RESOURCE DISCOVERY IN PEER-TO-PEER SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Beverly Yang
August 2004

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Andreas Paepcke

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom

Approved for the University Committee on Graduate Studies.

Abstract

In a peer-to-peer (P2P) system, distributed computing nodes of equal roles or capabilities exchange information and services directly with each other. At the application level, P2P systems have recently become a popular and effective means to pool together and harness large amounts of resources across organizations. Examples of existing and emerging applications include grid computing, file-sharing, and digital libraries. At the system level, due to their typically large scale, loose coupling and massive replication of resources, P2P systems offer potential strengths such as self-organization, self-healing, and robustness to failure or attacks. Because of these many strengths, P2P systems have become a topic of great interest in research and industry alike.

In a resource-sharing P2P system, many components interact to provide users with the ability to leverage the resources available in the system. For example, a scalable payment mechanism is required to handle transactions, while scheduling and resource allocation algorithms are needed by users to plan their budget. In this thesis, we focus on *resource discovery* – the key initial component required to allow effective location of desired resources across peers.

The design of a scalable and efficient P2P resource discovery mechanism presents several challenges in the areas of performance, participation and correctness. With regards to performance, in this thesis we present new algorithms for efficient query routing and data source selection, and study the most effective application of existing indexing techniques. With regards to participation, we propose an incentive-compatible query-forwarding protocol, and present an efficient micropayment scheme that may be used as a foundation for any economic application or resource discovery mechanism. Finally, with regards to correctness, we present possible attacks against our techniques and show the techniques to be robust. All our techniques are thoroughly evaluated over a file-sharing application model, which we derive from large-scale measurement experiments over live, operational P2P systems.

Acknowledgements

I would like to thank my advisor Hector Garcia-Molina for being a source of support and inspiration. His enthusiasm for new ideas and "interesting research questions" is contagious, and I learned much from observing his approach to research problems. Not only is Hector an excellent technical advisor, but he is also a warm person with a great sense of humor and a genuine concern for the well-being of his students. These past five years have gone by so comfortably and enjoyably, thanks to Hector.

I would also like to thank Jennifer Widom and Andreas Paepcke for kindly agreeing to be members of my reading committee, and for offering me sound advice and encouragement for years prior to the production of this thesis. I also thank Jennifer for introducing me to the world of research as an undergraduate.

I am grateful to all the members of the Stanford Database Group for their friendship and constructive comments. I would particularly like to thank my co-authors: Sriram Raghavan, Sergey Melnik, Neil Daswani, Patrick Vinograd, Sepandar Kamvar, Tyson Condie, and Prasanna Ganesan. I would also like to thank my officemates for making life at Stanford enjoyable: Neal Sample, Brian Cooper, Mor Naaman, and Zoltan Gyongi.

Finally, I would like to thank my wonderful family for their unconditional love and support. Throughout the years they have kept me going with encouraging discussions when things were difficult, good food when I was hungry, and happy times every time I saw them. They have never let me lose perspective or forget where my true joy and purpose in life lies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In a *peer-to-peer* (P2P) system, distributed computing nodes of equal roles or capabilities exchange information and services directly with each other. For example, in preservation systems, sites exchange storage resources to archive document collections. In grid computing systems, computers exchange available computing cycles. Recent advances in broadband communication, processing and disk technology have transformed commodity computers into valuable computing and storage nodes with efficient communication. As a result, over the past few years, the number of applications over P2P systems has grown tremendously.

Unlike traditional distributed systems, P2P systems are characterized by extremely large scale (e.g., millions of peers), lack of centralized control and a dynamic peer base distributed across wide area networks. Peers – typically personal computers or commodity servers – are highly unpredictable, unreliable, loosely-coupled nodes controlled by end-users belonging to different and often competing organizations.

By taking advantage of these unique characteristics (e.g., massive redundancy of resources, large scale, loose coupling), existing P2P applications exhibit potential strengths such as self-organization, self-healing, adaptation and fault tolerance. Perhaps most importantly, however, P2P applications have emerged as an effective and popular way to pool together and harness large amounts of *resources* such as data, compute cycles and knowledge/expertise across organizations. To illustrate, below we provide just a few examples of existing and emerging P2P applications:

- **Grid Computing** As described in [39], computational grids enable the sharing and aggregation of a wide variety of geographically distributed computational resources, and presents them as a single, unified resource for solving large-scale compute and data-intensive computing applications (e.g., molecular modeling for drug design, brain activity analysis and high energy physics). The idea is analogous to the electric power network (grid) where power generators are distributed, but the users are able to access electric power without bothering about the source of energy and its location. Unlike supercomputers or clusters managed by a centralized organization, nodes in a grid are managed autonomously, thereby complicating issues of resource management, discovery and compensation.

- **Web Services** The Web is quickly evolving from a large repository of static data into an environment where *services* are being offered – booking airline tickets, obtaining up-to-date news or financial information, managing personal bank accounts, etc. While accessing such services today requires that a human user invoke the service through a visual interface, the *web services* framework will allow businesses and organizations to publish services, and applications to invoke them automatically, without expensive and inefficient human intervention. *Semantic web services* [57] extend this vision with the automatic discovery, composition and execution of services to accomplish high-level tasks.

  Web services allow users and organizations to share expertise or business intelligence, while protecting intellectual property. However, as with grid computing, because services are owned by autonomous organizations, the management, discovery and purchase of services in a large-scale system becomes complex.

- **Data-Sharing** In the past few years, large-scale file-sharing systems (controversial due to the often illegal sharing of copyrighted files) have clearly demonstrated the ability of P2P systems to effectively pool together huge amounts of resources. Individual peers (typically users' PCs) share a controlled set of files stored on their file system, allowing other peers to download these files. By distributing the main costs of sharing data – disk space for storing files and bandwidth for transferring them – across peers in the network, these systems can scale easily to millions of users sharing petabytes of data, without the need for powerful, expensive servers.

- **Digital Archives** Digital archives are sites that preserve digital information over the long term (e.g., publishers and digital libraries). To preserve digital information, it is insufficient to simply make a few *local* backups, as backup tapes may break, publishers may go out of business, a local disaster may occur, and so on. Instead, through peer-to-peer data-trading protocols such as those proposed in [15], distributed digital archives can greatly increase the survival likelihood of their data by making copies at several geographically distributed archival sites.

- **Ubiquitous and Pervasive Computing** Computing nodes are becoming pervasive in every-day life, particularly with the proliferation of hand-held and wireless devices, as well as emerging "pervasive computing devices" – processing units embedded in cars, refrigerators, microwave ovens, etc. To provide the greatest value to the user, these devices and applications must communicate with each other to provide data and information into the temporal and spatial context in which the information is most useful [44]. The P2P framework emerges as a natural model of interaction between such devices, which often have unreliable communication (e.g., due to their mobile nature) and are controlled autonomously (e.g., the GPS systems of two friends' cars).

The ambitious and far-reaching goals of these P2P applications go far beyond what distributed systems do today. However, several key questions must be answered before this vision can be fully realized. For example, how can peers locate resources that they need? By what mechanism can peers make payments to one another? What incentives will peers have to contribute resources, or to behave correctly according to some protocol?

In this thesis, we will focus on one of the key unfinished pieces in P2P technology: *resource discovery*. To enable the effective exchange of services and information as described in each of these applications, users need a scalable, efficient tool by which to search for and locate desired resources. We will also show how designing a usable resource discovery mechanism requires addressing related challenges as well, such as questions of payment and incentives. The solutions in this thesis address both the core issue of resource discovery, and these related challenges.

Resource Discovery. Conceptually, resource discovery starts when a particular peer P has need of a resource R – for example, a particular file, or computation power, or a web service. Peer P gives the resource discovery system a *query message* containing a description of R, plus perhaps additional information such as how quickly R is needed, and how much P is willing to pay for the resource. The resource discovery system then produces a list L of peers capable of providing R (or some resource similar to R), and returns this list to P.

There are several general approaches to the resource discovery problem. One simple approach is to maintain a centralized index mapping resources to peers that provide them. Peers submit queries directly to the server hosting this index. Such an approach results in a *hybrid* P2P system, in which search is centralized, but resources and other functionality are distributed across peers.

Another potential approach is to have peers themselves route queries to each other. In such a *pure* P2P system, every node is a "servent" (both a client and a server), and can equally communicate with any other connected node. Peers form connections with other peers to create an *overlay network* – an application-level network built over the existing Internet routing substrate – and send messages through this overlay. For example, in the widely-used Gnutella [35] protocol, a peer P forwards its query for a resource R to its neighbors, who then forward the query to their neighbors, and so on. The query is "flooded" to a large number of peers in the overlay, who may then respond directly to P if they can provide R.

Pure systems may be further divided into *structured* and *unstructured* networks. In a structured overlay (e.g., [74, 78, 85]), both the data placement and overlay topology are tightly controlled. In contrast, the topology and placement of data or resources within an unstructured network are largely unconstrained. Search mechanisms over both types of overlays have their own strengths. For example, structured mechanisms can exploit the tight control over the peers in order to provide efficient queries with guarantees such as answering a query in a bounded number of messages. However, structured overlays also incur a relatively high maintenance cost as peers join and leave the system, can only support simple queries such as range [5] or lookup, and disrespect peer autonomy.

While each of the above approaches can be appropriate for different applications, in practice, hybrid and unstructured pure networks are more widely used by far. In this thesis we will explore techniques and algorithms towards resource discovery over these more widely used approaches. However, in Chapter 9 we discuss important future directions in both structured and unstructured networks.

Challenges. The same characteristics that make P2P systems so promising also present many technical challenges in the area of resource discovery. These challenges may be broadly classified into three categories – performance, participation and correctness.

With regards to *performance*, the large scale and decentralized nature of P2P systems allow them to share huge amounts of resources; however, locating a given resource R in the network becomes an expensive and time-consuming task. In a hybrid system, the load on the centralized index can be extremely high, while in pure systems, the number of messages passed among peers (e.g., in a flooding mechanism) may overwhelm the network. Many techniques previously developed for distributed systems of tens or hundreds of nodes cannot scale, and therefore no longer apply. Furthermore, load-balancing among peers is desirable for fairness and avoiding hotspots, but is difficult given decentralization.

Even the most efficient search algorithm is not useful without the *participation* of peers in the network. Typically in distributed systems, participation of nodes is a given, as the nodes belong to the same organization and strive towards a common goal. Because peers in a P2P system are commonly distributed across organizations, they may be selfish, and may not participate or cooperate with other peers. For example, peers may refuse to forward messages in a pure resource discovery mechanism, or they may refuse to contribute resources (e.g., compute cycles) to the system. The design of proper incentive mechanisms for participation is necessary in any system spanning organizations.

Similar to participation, the *correctness* of peer behavior is typically assumed in distributed systems, since participating nodes are trusted. Although it is possible for nodes to fail, they are generally not malicious. However, because a P2P system consists of peers from many unknown or untrusted organizations, peers may be victims of denial-of-service attacks during resource discovery, or they may receive inauthentic data or services in response to their request for some resource. Hence, the design of a P2P system must be robust against any kind of malicious, Byzantine peer behavior.

Clearly, the design of any good resource discovery mechanism must overcome the challenges outlined above. In this thesis, we address a number of these challenges; our contributions are described in detail in the next section.

1.1 Research Contributions and Thesis Outline

1.1.1 Efficiency

In the first half of this thesis, we study a number of different architectures and relevant techniques for efficient service discovery, in an environment where peers are cooperative. Three main themes have emerged from our work in this area. First, the techniques should be simple and practical enough to be easily incorporated into existing systems. Current successfully deployed P2P data-sharing systems all follow very simple protocols. Although these protocols are clearly suboptimal in efficiency, they highlight how simplicity is the key to wide and rapid adoption. Second, we need to understand the behavior of existing P2P applications. Effective search techniques need to make provisions for the unreliable nature of peers, and take advantage of observed user behavior. Finally, any technique should be adaptive, and tune itself according to the current state of the system. Because P2P systems are by nature highly dynamic, a rigid search mechanism that is effective in one scenario or for one particular user is likely to become ineffective as the system evolves or user changes.

Below we describe our contributions in this area, which follow the themes described above.

Characterization

To produce the proper algorithms and designs for P2P resource discovery, we must build upon careful characterization and analysis of P2P systems, which is in turn based on extensive experiments over live, large-scale systems. We need a profile of a "typical" application, such as the type of workload it encounters, the expected behavior of users and the common actions executed by the system. To this end, we present in Chapter 2 a model of a P2P file-sharing system, which we then use to evaluate our algorithms and protocols in following chapters. We validate this model over trace data gathered from live, large-scale P2P file-sharing systems.

Hybrid Search

Our initial work investigates search over the simplest P2P architecture: a hybrid system in which the shared resources or services are distributed across peers, but query processing is handled at a centralized index. Even in such a simple architecture, the scale of the system and unreliability of peers require one to make very careful design choices. In Chapter 3, we study the relevant issues and tradeoffs in designing a scalable hybrid P2P search mechanism. Using the system and query models developed in Chapter 2, we evaluate several hybrid P2P system architectures and discuss the tradeoffs among them, highlighting the effects of key parameter values on system performance.

Message Routing

In a pure P2P system, one of the keys to efficiency is a well-designed message routing protocol. In Chapter 4, we investigate simple but effective improvements over the existing, widely-used flooding protocol of systems such as Gnutella [35]. Chapter 4 presents the *Directed BFS* technique, which relies on feedback mechanisms to intelligently choose to which peer a message should be sent. Neighbors that have provided quality results in the past will be chosen first, yet neighbors with high loads will be passed over, so that good peers do not become overloaded. Chapter 4 also presents the *Iterative Deepening* technique, which allows search to proceed incrementally until peer P, who submitted the original query, has received a given number of responses. These two simple techniques allow search to be tuned on a per-query, per-user basis, and thus adhere to the design goals we outlined at the beginning of the section. Experiments over detailed query traces from the Gnutella network show that our techniques greatly reduce the cost of search, while maintaining good quality of results.

Role Differentiation

One of the well-known features of typical P2P systems is the heterogeneity of peers: some peers have much higher bandwidth, have greater computing capacity, or share more data than other peers. It is therefore natural to look for ways to exploit these differences among peers, and increase the efficiency of the system overall. One effective way to achieve this goal is to consolidate query processing responsibilities over a smaller number of peers. At one extreme, hybrid systems (Chapter 3) place the entire query processing load on a single server (or a very small number), while at the other extreme, pure systems completely distribute the load. In between these extremes lie the spectrum of *super-peer networks*. Super-peer networks differentiate between "super-peers" and "clients," where super-peers act as mini-index servers to a number of clients, but interact with each other as peers in a pure P2P system.

In Chapter 5, we investigate the proper design of super-peer networks that maximize overall efficiency while maintaining robustness and respecting individual peer capacities. Super-peers are used in currently deployed systems (e.g., KaZaA [51]) and have already proven to be effective in improving search performance. However, in this chapter we conduct an in-depth study on the design of super-peer networks and show how a straightforward implementation can be orders of magnitude less effective than one that is tuned to the particular requirements and workload of a system. From our investigation we present several design principles for an effective super-peer network, and a global design procedure that takes as input the workload and constraints on a system, and produces an efficient super-peer topology. Because workload and requirements evolve over time within a single system, it is important also to be able to evolve the design of the super-peer network to meet these changing needs. To this end, our global design procedure may be applied incrementally, such that peers can be directed to make runtime changes that tune the network to the current state of the system. We also present local decision-making guidelines, by which peers can make individual, runtime decisions that result in a globally efficient topology.

Non-Forwarding Search

The results of our studies in Chapter 4 show that incremental forwarding of query messages and intelligent server selection greatly improves search performance without affecting quality of results, while Chapter 5 shows that an improperly organized network topology and roles can result in high overhead in message forwarding and processing. These conclusions lead us to consider a new type of search architecture, in which messages are not forwarded, and peers have complete control over who receives their queries and when.

In Chapter 6, we study the *non-forwarding* architecture in the context of the GUESS protocol [40], an under-construction specification that is meant to become the successor of the widely-used but inefficient Gnutella protocol. Under the GUESS protocol, peers directly probe each other with their own query messages, rather than relying on other peers to forward the message. However, beyond this simple concept there are many issues to be addressed before the protocol can be successful. For example, when processing a query, in what order should peers be probed? The solution to this "server selection" problem must balance efficiency of the query with load-balancing among the peers. Also, if messages are not forwarded, then a peer must know of many other peers (rather than just a handful of neighbors) in order to successfully find answers to its queries. How should this large state be built up and maintained? In Chapter 6, we lay the foundations for studying non-forwarding protocols by identifying the key design decisions that affect system performance, and providing several simple *policies* for these decisions that achieve goals such as efficiency, fairness and robustness. We show how non-forwarding mechanisms can be over an order of magnitude more efficient than forwarding mechanisms like those studied in Chapter 4;

however, they must be deployed with care, as a naive implementation can result in highly suboptimal performance, and make them susceptible to hotspots and misbehaving peers.

1.1.2 Participation and Correctness

The second part of this thesis deals with incentives for peers to participate and behave correctly within the resource discovery mechanism. Our approach to dealing with these challenges is to use concepts from economics to construct a resource marketplace, where peers can buy and sell or trade resources as necessary.

Payments

The foundation of any economic system is a framework for efficient, secure payments. Payments may be used both as an incentive for peers to participate in resource discovery, or as a general mechanism for the buying and selling of resources. Because transactions in P2P applications are projected to be lightweight (i.e., involve payments of small amounts), and because vendor/buyer relationships tend to be unstable or transitory (due in part to the transitory nature of peers themselves), *micropayment* schemes such as [34, 58] appear to be the most appropriate approach to payment. A well-known problem with most existing micropayment schemes is the heavy load on a trusted, centralized broker, which is required to handle accounts, distribute and cash coins and provide security. In these schemes, broker load is generally $O(n)$ in the number of transactions. Brokers therefore present a scalability and performance bottleneck, and it is unclear whether they can scale to the size dictated by P2P systems. In Chapter 7, we present PPay, a micropayment scheme that exploits unique characteristics of P2P systems to maximize efficiency while maintaining security properties. We show how the basic PPay protocol far outperforms existing micropayment schemes, while guaranteeing that all coin fraud is detectable, traceable and unprofitable.

Message Routing with Incentives

Competitive P2P networks are applications in which peers are in competition with each other to provide services. Examples of such applications include pay-per-transaction file-sharing, auctions and web services. Data and service discovery mechanisms in such applications are threatened by the *non-cooperation* problem, where peers do not forward queries to potential competitors. In Chapter 8, we show how non-cooperation causes unacceptable degradation in the quality of results achieved by search mechanisms designed for cooperative environments. We then present an economic protocol to address this problem. The protocol, called the *RTR* protocol, is based on the buying and selling of the right-to-respond (RTR) to each query in the network. Through simulations we show how the RTR protocol not only overcomes non-cooperation by providing the proper incentives to peers, but also results in a network that is even more effective *and* efficient through intelligent, incentive-compatible routing of messages.

1.2 Related Work

In this section, we review work that is broadly related to the topics in this thesis. Specific related research is cited in individual chapters.

The original motivation for much P2P research was early file-sharing P2P systems such as Gnutella [35], Napster [62], Freenet [30] and Kazaa [51]. Napster used a hybrid search architecture containing some centralized components. Gnutella and Freenet were pure networks; Freenet used a sequential depth-first search to discover files, while Gnutella employed a flooding-based message routing protocol to disseminate queries. Gnutella and Kazaa both introduced the super-peer architecture at roughly the same time. The *free-riding* phenomenon – in which a huge fraction of peers share no files, and only a small fraction handle the majority of the queries – was one of the early problems caused by poor incentive design, and was first noticed in Gnutella [2]. While these applications were among the first to demonstrate the possibilities of P2P systems, they were also controversial due to illegal sharing of copyright files. Since then, attention has been turned to legitimate, business-oriented applications such as grid computing and web services.

Outside of message routing and role differentiation, studied in Chapters 4 and 5, respectively, many other techniques may be used to improve search performance over unstructured networks. Examples include replication or caching of data (e.g., [12, 30, 92]), adaptive topologies (e.g., [11, 16, 66]), matching topologies to the underlying network (e.g., [32, 78]) and query caching (e.g., [87]).

Our work in incentives is inspired by the emerging field of *algorithmic mechanism design* (AMD) (e.g., [24, 27, 68, 69, 72]). AMD has been successfully applied to issues such as optimal routing and resource allocation [26, 54]. Recent activity in the field of AMD has focused on P2P networks [28, 84], since the autonomous nature of peers makes proper incentives crucial. However, while we apply many ideas from AMD and classic game theory, our problem cannot be fully described by the existing tools. For example, there is no precise way of capturing the adaptive nature of the network topology. More successful applications of AMD to P2P systems have typically focused on the simpler problem of file-sharing incentives (e.g., [36]).

Chapter 2

P2P Application Model

In order to properly evaluate the effectiveness of a resource discovery mechanism, we need to understand the applications over which such a mechanism would be used. In particular, we must have a characterization of workload, user behavior and system actions. In this chapter, we lay the foundation for our research by developing a realistic, parameterized model of a P2P file-sharing system. This model is then used in later chapters as an evaluation tool for our algorithms and techniques.

Because file-sharing was the most widespread early P2P application, almost all traceable data from large-scale, live P2P systems come from these file-sharing systems. As a result, we believe file-sharing is the most suitable model application for our studies, as our models can be validated against a wealth of empirical data, and user behavior in such applications can be well-understood. From this solid starting point, we can then modify the parameters of our model to project different types of applications.

We first review the basic actions in a file-sharing system in Section 2.1, and describe in Section 2.2 the live file-sharing applications that we studied and from which we gathered general usage statistics. We then develop a probabilistic model to describe query behavior and expected query result sizes in Section 2.3. In Section 2.4, we develop an analytic model to describe the cost of actions in the P2P system. We validate all of our models using trace data collected from running, publicly available P2P systems.

We note that P2P systems are complex, so the main challenge is in finding query and performance models that are simple enough to be tractable, yet faithful enough to capture the essential tradeoffs. While our models (described in Sections 2.3 and 2.4) contain many approximations, we believe they provide a good and reliable understanding of both the characteristics of P2P systems, and the tradeoffs among the architectures. Sections 2.3.2 and 2.4 discuss in further detail the steps we took to validate our models.

We also note that the query and performance models and usage statistics are not a complete simulation model in themselves, but are building blocks that can be used to create a realistic simulation. Hence in every chapter, we will still describe the specific simulation steps used to evaluate each technique, and how the models presented in this chapter are used within the simulation.

Many of the contributions of this chapter were first published in [94].

2.1 File-Sharing Overview

In a file-sharing system, users specify a number of files, residing in their file systems, which they are willing to share. A "shared" file is one that other users in the system are allowed to download.[1]

Every peer in the network is conceptually controlled by a user, although in practice, a single human user may control many peers, or a peer may be automated. For clarity, our discussions will adhere to this conceptual model, and assume there is a one-to-one correspondence between users and peers. For simplicity, we may sometimes use the term "peer" to describe the conceptual human user controlling the peer (e.g., we may say that a peer is sharing a number of files, when it is really the user sharing files).

In addition, every peer is conceptually a distinct node in the network, although in practice, a single machine with a single network interface may be running multiple peers. In this thesis, we will use the terms "node" and "peer" interchangeably. The only exception is when we discuss *hybrid* systems, in which a node may also refer to a centralized server (see Chapter 3).

Three user-level actions are supported in a file-sharing system: *join/leave*, *query* and *update*. The lower-level actions taken by the peers for each of these user actions differs according to the type of index, role differentiation and message routing used by the particular resource discovery mechanism. While details of different mechanisms are discussed in later chapters, here we will use a few examples to illustrate the purpose of each user-level action. Figures 2.1 to 2.3 present a visual illustration of each type of action.

When a user joins the system, her file *collection* is available for other users to download.

[1] Access control is typically supported, but we do not model this aspect of the application.

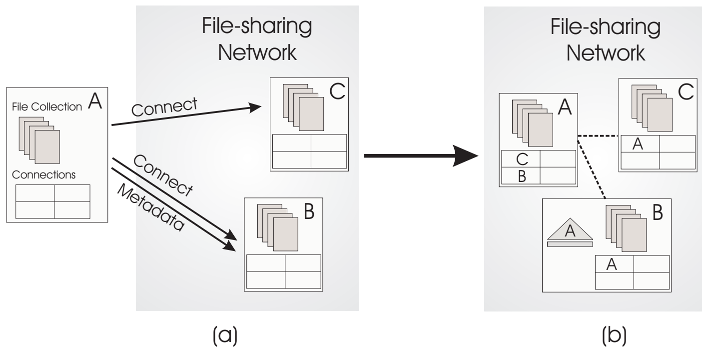

Figure 2.1: (a) Peer A joins the system by connecting to peers B and C. A also sends metadata to B. (b) After the join completes, A has B and C in its connection table (and vice versa), and B has an index on A's collection.

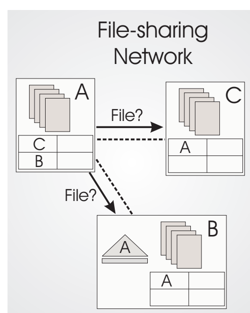

Figure 2.2: Peer A forwards its query to neighbors B and C.

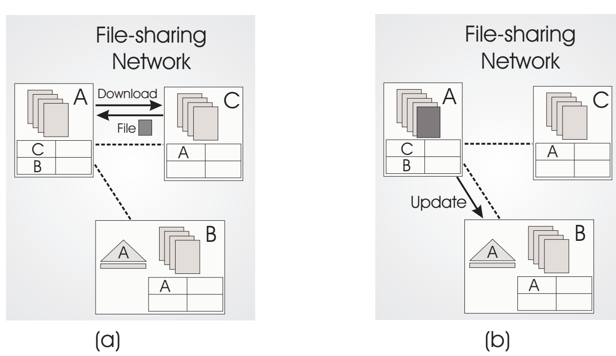

Figure 2.3: (a) Peer A requests a download from C, who then sends the requested file to A. (b) After the download completes, A notifies B of the update, so that B can modify its index of A's content.

If an index of the user's files needs to be built at a remote node (e.g., at a remote peer or a centralized index), then the user's peer must export *metadata* describing the collection to that remote node. For example, in Figure 2.1, peer A joins the system, and exports its collection metadata to peer B, who will build an index over A's collection. This metadata might include file names, creation dates and copyright information. For now, we assume that the index takes the form of inverted lists [81]. Every file's metadata is considered a document, with the text of the file name, author name and so on. Similarly, when a user leaves the system, the remote node may remove that user's collection information from its index.

Regardless of whether an index is built at remote nodes, a peer must be connected to other nodes in the network, whether they be other peers, or centralized servers in a hybrid system. Nodes therefore maintain a table of *connection information*, describing active connections (e.g., IP address, line speed, etc). When a peer joins the system, its *neighbors*, to whom it makes connections, will modify their connection information table. For example, in Figure 2.1 when peer A joins the network and connects to peers B and C, both B ad C will update their connection information.

In order to discover files, a user will submit a query via her peer. A *query* consists of a list of desired words to appear in a file's metadata. Depending on the search mechanism in use, the peer may take a number of actions, such as forwarding the query to a number of other peers it is connected to (as illustrated in Figure 2.2), or sending the query to the hybrid server. When a node receives a query, it may choose to process it, drop it and/or forward it. When a peer or hybrid server processes a query, it searches for matches in its index. A query is processed by retrieving the inverted lists for all its words, and intersecting the lists to obtain the identifiers of the matching documents (user files). Results, typically pointers to or metadata of files, are then returned back to the querying peer. Often, users will specify the number of results they wish returned. A query is said to be *satisfied* if at least this number of results are returned.

The user examines the query results, and when she finds a file of interest, her peer directly contacts the peer holding the file, and *downloads* the file. After a peer P successfully downloads a file, if there are any remote nodes indexing P's collection, P will need to notify these remote nodes of this *update*. Those remote nodes will then add this information to their index. Similarly, if files are added or deleted from P's collection through other means, these updates must be propagated to the remote nodes indexing P's collection. This process

Name	Description	OpenNap	Gnutella
FilesPerUser	Average files shared per user	168 files	340 files
RecordSize	Average size (in bytes) of result record	50 B	76 B
MetadataSize	Average size (in bytes) of metadata for a single file	50 B	72 B
WordsPerMetadata	Average number of words used to describe a file in its metadata	10	–
WordsPerQuery	Average number of words used in a query	2.4	–
QueriesPerUserSec	Rate of queries per user per second	$8.33 \cdot 10^{-4}$	$9.26 \cdot 10^{-3}$
QueryJoinRatio	Ratio of queries to joins	0.45	–
QueryDownloadRatio	Ratio of queries to downloads	0.5	–
FracChange	Fraction of user's collection changed offline	.1	–
ActiveFrac	Percent of the total user population that is active at any given time	.05	–

Table 2.1: General Statistics

is illustrated in Figure 2.3, in which peer A decides to download a file from peer C, and notifies peer B of this action so that B may update its index over A's collection.

2.2 System Usage

In this section, we briefly describe the experiments we ran over OpenNap [70] and Gnutella – two live, open file-sharing P2P systems – and the general usage statistics we gathered from both these systems, shown in Table 2.1. We also ran validation experiments and gathered trace data over OpenNap and Gnutella; these activities will be described later in the relevant sections and chapters.

OpenNap. The OpenNap project is open source server software that clones Napster, originally one of the most popular examples of file-sharing hybrid P2P systems. An OpenNap server is simply a host running the OpenNap software, allowing peers to connect to it and offering the same file lookup, chat and browse capabilities offered by Napster. In fact, the protocol used by OpenNap follows the official Napster message protocol, so any peer following the protocol, including actual Napster peers, may connect. Like Napster, OpenNap was used almost exclusively for sharing music files. OpenNap servers can be run independently, or in a "chain" of other servers that communicate with each other to form a single "virtual" server from the peers' perspective (see Chapter 3 for more details). All users connected to a server can see the files of local users on any of the other servers in the chain (a capability Napster lacks).

Our data comes from an OpenNap server run in a chain of 5 servers. Up to 400 simultaneous real users connected to our server alone, and up to 1350 simultaneous users were connected across the entire chain. Once an hour, the server logged workload statistics such as the number of queries, joins and downloads occurring in the past hour, length of query strings and the current number of files in the server. These usage statistics are shown in the OpenNap column of Table 2.1. All statistics were completely anonymous to protect the privacy of users. The data was gathered over an 8 week period, in August and September 2000, and only data from the last six weeks was analyzed, to ensure that the data reflected steady-state behavior of the system. Naturally, the workload on our server varied throughout the experimental period depending on the time of day and day of week. However, the variation was regular and predictable enough such that we could average the numbers to get good summary information to work with.

While most of the statistics shown in Table 2.1 are straightforward, a few of them deserve further comment. First, our measured value of *QueryJoinRatio* was 0.45, meaning roughly that on average users submit one query every other session. We were surprised by this number, as we expected users to at least submit one query per session. After analyzing our data, we discovered two main reasons for the low *QueryJoinRatio* value. First, many peers operated over unreliable modem connections. If the connection goes down, or is very slow, the peer will attempt to re-login to the server. A peer may thus join several times in what is a single session from the user's point of view. Second, many users are drawn to the OpenNap chat community. Users often log on to the server simply to chat or see who is online, rather than to download files. Our data shows that chat accounts for 14.6 KB/hour of network traffic – not enough to significantly impact the system performance, but enough to bias *QueryJoinRatio*. For our evaluation we initially use the measured value of 0.45, but we also consider higher values that may occur with more reliable connections or with less chat-oriented systems.

Parameter *FracChange* represents the fraction of a user's file collection that changes between a leave and the next join. Since in OpenNap one cannot relate a new join to a previous leave, this value cannot be measured directly. A file collection may change offline because a user deletes files, or because she "rips" a CD. We estimate that *FracChange* = 10% of a user's 168 average number of files may be changed, mainly because files that were downloaded are played a few times and then removed to make room for new material. In later chapters (e.g., Section 3.5) we perform a sensitivity analysis on parameter *FracChange* to see how critical our choice is to system performance.

Parameter *ActiveFrac* indicates what fraction of the user population is currently active. We estimate that $ActiveFrac = 0.05$ as follows. Our statistics show an average of 226 active local users at any given time, and with $QueryPerUserSec/QueryJoinRatio = .00184$ joins per user per second, there are approximately 36015 joins a day. Let us assume that the average user submits 3 to 4 queries a day. This means the average user logs in $QuerysPerUserPerDay/QueryJoinRatio = 8$ times a day. (The high value for joins per user per day is a direct result of the low value observed for *QueryJoinRatio*. See discussion above.) If each user logs on 8 times a day, then there are a total of approximately 4502 users in our local user base, which means on average, approximately $226/4502 = .050$ of all users are active at any given time. Again, in Section 3.5 we study the impact of changing this parameter.

Gnutella. The Gnutella network is the largest open pure P2P network in operation, with 500,000 users as of May, 2001. We therefore chose to base our evaluations of pure systems on data gathered from this network.

General usage statistics for Gnutella are also shown in Table 2.1. To gather these statistics over Gnutella, for a period of one month in August and September 2001, we ran a single Gnutella client that observed messages as they passed through the network. Based on the content of these messages, our client could determine characteristics of users' collections, and of the network as a whole. For example, Gnutella Pong messages contain the IP address of the node that originated the message, as well as the number of files stored at the node. By extracting this number from all Pong messages that pass by, we can determine the distribution of collection sizes in the network. These statistics are also listed in Table 2.1.

We ran many additional experiments over the Gnutella network, in which our client actively spawned messages and interacted with other peers. These experiments are described further in Chapter 4.

Discussion. A server in a hybrid system has a global view of all users' activity, while a single Gnutella client has a limited perspective of the network. Therefore, a number of statistics could not be gathered in Gnutella (such as the ratio of queries to joins). In such cases, when running our experiments in later chapters, we will borrow the statistics observed in OpenNap.

We also note that some characteristics in Gnutella appear to be very different from those in OpenNap. For example, the rate of queries per user is over 10 times higher in Gnutella than in OpenNap. One possible explanation is that again, our Gnutella client only had a limited view of the system. Since our client was well-connected and ran over a high-bandwidth connection, it was likely part of the "core" of the network, where the most active peers resided. As a result, our client saw more activity than the average peer would. Later performance studies produced elsewhere (e.g., [41, 82]) ran much larger and more thorough experiments, in which many Gnutella clients were connected at multiple locations in the network. Therefore, we often use the statistics generated in these external performance studies where applicable, and clearly cite the instances in which we do so.

2.3 Query Model

To compute the cost of processing queries, we need a way to estimate the number of query results, and the expected number of nodes that will have to process a query. In this section we describe a simple query model that can be used to estimate these values.

We assume a universe of queries $q_1, q_2, q_3, \ldots$. We define two probability functions over this universe:

- g – the probability density function that describes query popularity. That is, $g(i)$ is the probability that a submitted query happens to be query q_i.
- f – the probability function that describes query selectivity. In particular, if we take a given file in a user's library, with probability $f(i)$ it will match query q_i.

For example, if $f(1) = 0.5$, and a library has 100 files, then we expect 50 files to match query q_1. Note that distribution g tells us what queries users like to submit, while f tells us what files users like to store.

2.3.1 Modeling f and g

While we can use any f and g distributions in our model, we have found that exponential distributions reflect well the actual behavior of queries and responses (see Section 2.3.2) and are computationally easier to deal with. Thus, we assume for now that $g(i) = \frac{1}{\lambda_g} e^{-\frac{i}{\lambda_g}}$. Since this function monotonically decreases, this means that q_1 is the most popular query, while q_2 is the second most popular one, and so on. The parameter λ_g is the mean. If λ_g

is small, popularity drops quickly as i increases; if λ_g is large, popularity is more evenly distributed.

Similarly, we assume that $f(i) = \frac{1}{\lambda_f} e^{-\frac{i}{\lambda_f}}$. Note that this assumes that popularity and selectivity are correlated. In other words, the most popular query q_1 has the highest selectivity, the second most popular query q_2 has the second highest selectivity and so on. This assumption is reasonable because in a "library-driven" system where queries are answered over users' personal collection of files, popular files are *queried* for frequently, and *stored* frequently.[2] Stored files can be obtained from the system by downloading query results, or from an external source. (For example, in the music domain, files can be obtained and stored from "ripped" CDs, and we expect a correlation between these files and what users are looking for.) However, the mean λ_f can be different from λ_g. For example, if λ_g is small and λ_f is large, popularity drops faster than selectivity. That is, a rather unpopular query can still retrieve a fair number of results.

Some readers may be concerned that, under these assumptions, eventually everyone would have stored the popular files, and no one would need to query for them anymore. However, at least in the music domain (in which many file-sharing applications belong), such a steady state is never reached. One reason is that users do not save all files they search for, or even all files they download. In particular, a study done by [75] shows that on average, college-age users who have downloaded over 10 songs eventually discard over 90% of the downloaded files. Hence, the high frequency of a query does not necessarily drive up the size of the result set returned. Another reason why a steady state is never reached is that user interests (and hence the ordering of queries by popularity and selectivity) vary over time. Our model only captures behavior at one instant of time.

From this point on, we will express λ_g as $r \cdot \lambda_f$, where r is the ratio λ_g/λ_f. For a given query popularity distribution g, as r decreases toward 0, selectivity f becomes more evenly distributed, and queries tend to retrieve the same number of results regardless of their popularity.

2.3.2 Calculating Responses

Given any definition for functions f and g, we can calculate a number of useful values:

- $M(n)$, the expected number of results returned from a collection of n files.

[2] This characteristic also holds true for "cache-driven" systems like Freenet that use the LRU replacement strategy.

- $M'(n, R)$, the expected number of results returned from a collection of n files, when the maximum number of results allowed is R.
- $Q(n, R)$, the probability that R or more query matches can be found in a collection of n or fewer files.
- $K(n, R, S)$, the expected number of nodes that must process a query in order to return at least R results, given that each node processes the query over a collection of n files. We assume that only S total nodes exist in the network.

Calculating $M(n)$

First, we make the simplifying assumption that the probability that a given file matches a given query is independent of the probability that other files in the same collection match the query. This assumption is especially true, for example, in a hybrid system where each server indexes a very large collection.

For query q_i, the probability of retrieving m results in a collection of n files is the same probability of getting m successes in n Bernoulli trials, where the probability of success is $f(i)$. The expression for the probability is:

$$B(n,m) = \binom{n}{m} (f(i))^m (1 - f(i))^{n-m}. \tag{2.1}$$

Because the probability of a success depends on which query is considered, we must take the expected value of this probability. Hence, $T(m,n)$, the expected value for $B(n,m)$, can be computed as:

$$T(n,m) = \sum_{i=0}^{\infty} g(i) \left(\binom{n}{m} (f(i))^m (1 - f(i))^{n-m} \right). \tag{2.2}$$

Finally, since the number of results can range from 0 to n, the expected number of results returned for a query is:

$$M(n) = \sum_{m=0}^{n} m \left(\sum_{i=0}^{\infty} g(i) \left(\binom{n}{m} (f(i))^m (1 - f(i))^{n-m} \right) \right). \tag{2.3}$$

Calculating $M'(n, R)$

In some cases, it is useful to cap the number of results returned at some value R. That is, even if a very large number of results are present, only R will be returned. In such a scenario, the expected number of results returned changes.

For a given query q_i, the expected total number of results from a collection of n files, given a cap R, is:

$$\sum_{m=0}^{R-1} B(n,m) \cdot m + \sum_{m=R}^{n} B(n,m) \cdot R \tag{2.4}$$

where $B(n,m)$ is defined earlier as the probability of getting m successes in n trials, and where the probability of a success is $f(i)$ – that is, the probability of getting m results in a collection of n files. Note that in the second term of the expression, if R or more results are found for a query, only R results are returned.

Equation 2.4 can be rewritten as:

$$\begin{aligned} \sum_{m=0}^{R-1} B(n,m) \cdot m + \sum_{m=0}^{n} B(n,m) \cdot R - \sum_{m=0}^{R-1} B(n,m) \cdot R \\ = R + \sum_{m=0}^{R-1} B(n,m) \cdot (m - R) \\ = R - \sum_{m=0}^{R-1} B(n,m) \cdot (R - m). \end{aligned} \tag{2.5}$$

Since the probability of a success in the above expression depends on which query is considered, we must take the expected value of this expression over all queries. Hence, $M'(n, R)$, the expected value of the above expression, is computed as:

$$M'(n,R) = \sum_{i=0}^{\infty} g(i) \Big(R - \sum_{m=0}^{R-1} B(n,m) \cdot (R - m) \Big). \tag{2.6}$$

Calculating $Q(n, R)$

To compute $Q(n, R)$, the probability that R or more query matches can be found in a collection of n or fewer files, we use $T(n, m)$, defined earlier as the probability of finding exactly m answers in a collection of exactly n files. $Q(n)$ can be restated as the probability

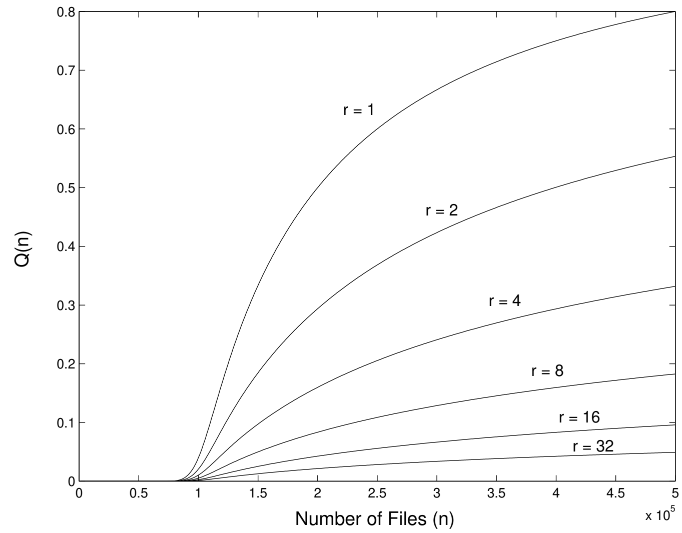

Figure 2.4: Q(n) distributions for $r \geq 1$

that we do not get $0, 1, ...$ or $R-1$ results in exactly n files, or

$$Q(n) = 1 - \sum_{m=0}^{R-1} T(n,m). \tag{2.7}$$

Figure 2.4 illustrates $Q(n)$ for several cases where $r \geq 1$. For this graph, $R = 100$, and $\lambda_f = 1000$. We can see that if there are few files, then it is very unlikely that the desired $R = 100$ results will be found. However, as the n increases (e.g., beyond a hundred thousand), the probability $Q(n)$ increases rapidly. As r decreases, λ_g decreases and the popularity distribution gets more skewed. The very popular queries are more likely to occur, and they get more results, so the threshold R is reached sooner. Hence, curves for smaller r rise more steeply.

Calculating $K(n, R, S)$

Let $P(s, n, R)$ represent the probability that exactly s nodes, out of an infinite pool, are needed to return R or more results, when each node indexes a collection of n files.

The expected number of nodes is then:

$$K(n, R, S) = \sum_{s=1}^{S} s \cdot P(s, n, R) + \sum_{s=S+1}^{\infty} S \cdot P(s, n, R) \tag{2.8}$$

where S is the number of nodes in the actual system. (The second summation represents the case where more than S nodes from the infinite pool were needed to obtain R results. In that case, the maximum number of nodes, S, will actually be used.)

Now, let us define $L(s,n,R)$ to be the probability that s *or fewer* nodes are needed to return R or more results. Then, $P(s,n,R) = L(s,n,R) - L(s-1,n,R)$, and $L(0,n,R) = 0$ and $L(\infty,n,R) = 1$. The above expression for $K(n,R,S)$ can be rewritten as:

$$K(n,R,S) = \sum_{s=1}^{S} s\cdot(L(s,n,R)-L(s-1,n,R)) + \sum_{s=S+1}^{\infty} S\cdot(L(s,n,R)-L(s-1,n,R)). \quad (2.9)$$

The first expression in Equation 2.9 can be expanded as follows:

$$\begin{aligned}
&\sum_{s=1}^{S} s\cdot(L(s,n,R) - L(s-1,n,R)) \\
&\quad = S\cdot L(S,n,R) - S\cdot L(S-1,n,R) + (S-1)\cdot L(S-1,n,R) - (S-1)\cdot L(S-2,n,R) \\
&\qquad ...2\cdot L(2,n,R) - 2\cdot L(1,n,R) + L(1,n,R) - L(0,n,R) \\
&\quad = S\cdot L(S,n,R) - \sum_{s=1}^{S-1} L(s,n,R). \qquad (2.10)
\end{aligned}$$

Similarly, the second expression in Equation 2.9 can be expanded as follows:

$$\begin{aligned}
&\sum_{s=S+1}^{\infty} S\cdot(L(s,n,R) - L(s-1,n,R)) \\
&\quad = S\cdot(L(S+1,n,R) - L(S,n,R) + L(S+2,n,R) - L(S-1,n,R) \\
&\qquad + L(S+3,n,R) - L(S-2,n,R)...) \\
&\quad = S\cdot L(\infty,n,R) - S\cdot L(S,n,R) \\
&\quad = S\cdot 1 - S\cdot L(S,n,R) \qquad (2.11)
\end{aligned}$$

Putting the two expressions together, we have:

$$\begin{aligned}
&K(n,R,S) \\
&\quad = S\cdot L(S,n,R) - \sum_{s=1}^{S-1} L(s,n,R) + S - S\cdot L(S,n,R) \\
&\quad = S - \sum_{s=1}^{S-1} L(s,n,R). \qquad (2.12)
\end{aligned}$$

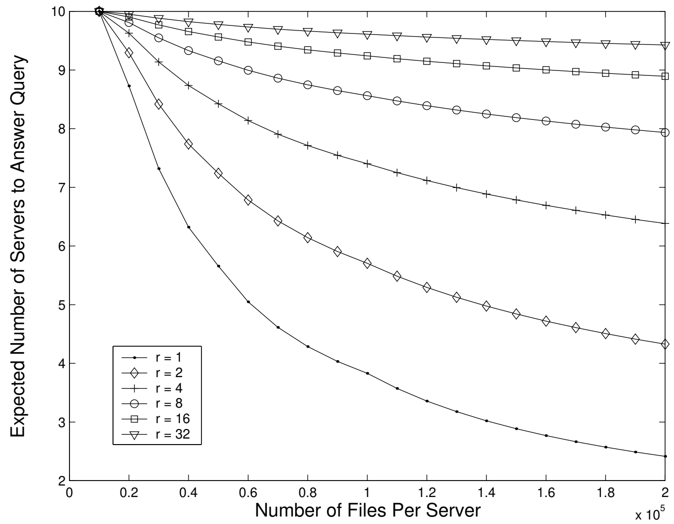

Figure 2.5: $K(n, R, S)$ for $r \geq 1$

Now, let $Q(n, R)$ be the probability that R or more results will be returned from a collection of n or fewer files. Then, the probability that s or fewer nodes are needed to return R or more results, $L(s, n, R)$, is equal to the probability that R or more results will be returned from a collection of at most as many files as are stored in s nodes. That is, $L(s, n, R) = Q(s \cdot n, R)$. Therefore,

$$K(n, R, S) = S - \sum_{s=1}^{S-1} Q(s \cdot n, R) \tag{2.13}$$

To illustrate our model, Figure 2.5 shows values for $K(n, R, S)$ as a function of the number of files *per node*. In this example, we model a hybrid system in which 10 large servers process all queries, and the number of files covered by each server is roughly the same. We assume $R = 100$ (the default cap value used by several real-life systems), and $\lambda_f = 1000$. We vary the value of r across the different curves, and the number of files covered by each server across the x-axis. When the number of files per server, n, is small (e.g., less than 20,000), then a query will be serviced by all 10 servers in order to attempt to reach the threshold of R results (there is no guarantee that even in 10 servers, all R results will be found). As n grows, however, it becomes more likely to find R results in a smaller number of servers, hence $K(n, R, S)$ decreases. For example, we see in the figure that if each server holds 200,000 files, and $r = 1$, then the query need only be serviced by

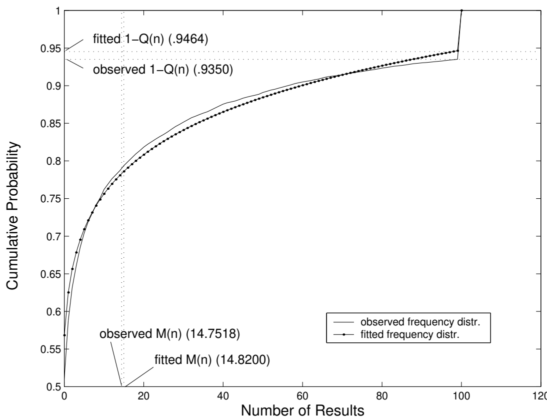

Figure 2.6: Observed and fitted frequency distributions of results in OpenNap

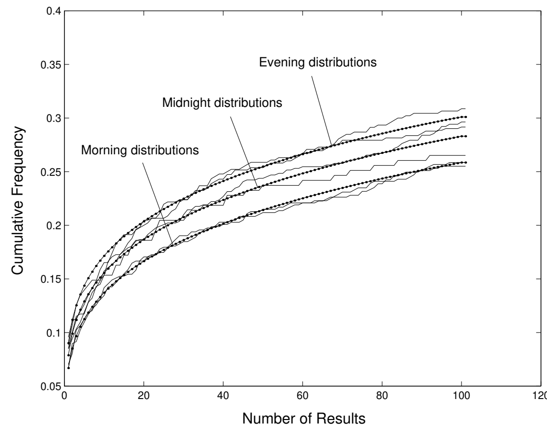

Figure 2.7: Observed and fitted frequency distributions of results in Napster

2.4 servers on average before R results are found. Note that as r decreases, the threshold R is reached sooner because queries with high selectivity are more likely to be queried. As a result, $K(n, R, S)$ is lower.

Validation

We validated our query model over data gathered from the OpenNap hybrid file-sharing system. As described in Section 2.2, we controlled one of the index servers in an OpenNap system. Each hour, the server would output an hourly log that recorded how many queries in the 1-hour period obtained 0 results, how many obtained 1 result, and so on. In effect, for every hour we obtained a histogram giving the frequency at which each number of results was returned by the server. The log also recorded the number of files covered by the index stored at that server for that hour. The observed number of files ranged between 40,000 and 95,000 files. Clearly, the histogram depended on the number of files; when there were more files, users tended to get more results.

To increase the accuracy of our measurements, we combined 19 histograms that occurred when the server had roughly 69,000 files (actually between 64,000 and 74,000 files). Thus, the combined histogram showed us how many queries in all the relevant periods had obtained 0 results, how many had obtained 1 result, and so on. Figure 2.6 shows the cumulative normalized version of that histogram (solid line). For example, about 87.3% of all the queries had 40 or fewer results. At $R = 100$ results we observe a jump in the distribution because at most 100 results are returned by the server. Thus, 93.5% of the queries returned 99 or fewer results, and $1 - 93.5\% = 6.5\%$ of the queries would have returned 100 or more results, but were forced to return only 100. Keep in mind that Figure 2.6 shows results for a single server. When a query returns less than 100 local results, other servers in the system may have been contacted for more results, but those results are not shown in this graph.

Our next step is to fit the curve of Figure 2.6 to what our query model would predict. Since $T(n, m)$ is the probability that a query returns m results when run against n files (see Section 2.3.2), the cumulative distribution of Figure 2.6 should match the function $h(x) = \sum_{m=0}^{x} T(69000, m)$ in the range $0 \leq x \leq 99$. By doing a best fit between this curve and the experimental data, we obtain the parameter values $\lambda_f = 400$ and $r = 10$ (and $\lambda_g = 400 \cdot 10 = 4,000$). These parameters mean that the 400th query has the mean selectivity power, while the 4000th query has the mean probability of occurring. Figure 2.6 shows the fitted curve $h(x)$ with $\lambda_f = 400$ and $r = 10$.

With the fitted parameters, our model predicts a mean number of results, $M'(69000, 100)$, of 14.82, which differs from the observed mean by only 1.22%. Similarly, our model predicts that with probability $1 - Q(69000, 100) = 0.9464$, there will be fewer than 100 results. This only differs by 0.46% from the measured value (see Figure 2.6). Thus, our model is clearly able to describe well the observed data from our OpenNap experiments.

To observe how our model worked in a different setting, we ran a second set of experiments. We took a random sample of 660 queries submitted to the OpenNap server and submitted them to Napster. We submitted the same sample of queries six times at different times of day – evening (7 pm), midnight (12 am) and morning (8 am), when the load on Napster was heaviest, lightest and average, respectively. We recorded the number of results returned for the queries, obtained a result frequency distribution for each run, and fitted curves to the observed data.

Table 2.2 shows the results for these experiments, together with the λ_f and r values obtained by curve fitting. As expected, the values for $M(n)$ and $1 - Q(n)$ are closely matched by the fitted values within each run, and parameters r and λ_f are fairly close across the runs. Interestingly, the obtained λ_f and r values do not only differ from the values we obtained from OpenNap, but they also differ slightly by time of day. Note for example that, even though more files are available in the evening experiments, fewer average

Run	Number of Files	Time of Day	$M(n, 100)$	$1 - Q(n, 100)$	λ_f	r	Fitted $M(n, 100)$	Fitted $1 - Q(n, 100)$
1	1335411	morning	80.01	.2585	100	3.68	79.44	.2633
2	1539166	morning	80.03	.2550	100	3.68	80.21	.2538
3	1969233	evening	75.77	.3087	140	4.14	75.86	.3025
4	2038807	evening	76.33	.2963	140	4.14	76.06	.2999
5	1256387	midnight	77.38	.2917	100	3.92	77.21	.2890
6	1468911	midnight	78.17	.2830	100	3.90	77.68	.2830

Table 2.2: Characteristics of Napster Query Runs

results are obtained by the queries (smaller M values). The tight correlation between query characteristics and time of day is highlighted in Figure 2.7, where the observed Napster distributions are marked in solid lines, and the fitted distributions in dotted lines.[3]

The explanation for the differences in parameters between the two systems, and between the different times of day, is that the communities that use P2P systems vary, and hence the types of files they make available (and the queries they submit) differ. Morning Napster users are not at school and wake up early, and have somewhat different interests from Napster users at midnight, who are more likely to be college students. Hence, morning and midnight query behavior differ. Similarly, since Napster is widely known, it tends to have more casual users who access popular songs. OpenNap, on the other hand, is harder to find and use, so its users tend to be more involved in the music-sharing concept and community, more technically savvy, and may not always have tastes that conform to the mainstream. As a result, query frequencies and selectivities for OpenNap are more evenly distributed, accounting for the larger λ_f and larger r. In contrast, the popularity skew of songs in Napster is probably high and heavily influenced by the latest trends. Hence, λ_f and r are relatively small.

We conclude from these observations that our query model describes the real world sufficiently well, although we may need to consider the effect of different λ_f and r values on the performance of the various architectures.

2.4 Performance Model

To evaluate the cost of actions incurred by a given resource discovery mechanism, we need to determine the cost of the three user-level actions described in Section 2.1: join/leave,

[3]There is again a jump at $R = 100$ results to cumulative frequency of 1, but it is not shown because the value 1 is off the scale.

Action	Bandwidth Cost (Bytes)			Processing Cost (Units)
	Napster	Gnutella	GUESS	
Send Query	100 + query	82 + query length	70 + query length	$.44 + .003 \cdot$ query length
Recv. Query	100 + query length	82 + query length	100 + query length	$.57 + .004 \cdot$ query length
Process Query	0	0	0	$14 + 1.1 \cdot \#$ results
Send Response	$140 \cdot \#$ results	$80 + 28 \cdot \#$ addr $+76 \cdot \#$results	$208 + 76 \cdot \#$results	$.21 + .31 \cdot \#$ addr $+.2 \cdot \#$results
Recv Response	$140 \cdot \#$ results	$80 + 28 \cdot \#$ addr $+76 \cdot \#$results	$208 + 76 \cdot \#$results	$.26 + .41 \cdot \#$ addr $+.3 \cdot \#$results
Send Join	$42 + 125 \cdot \#$ files	$80 + 72 \cdot \#$ files	n/a	$.44 + .2 \cdot \#$ files
Recv. Join	$42 + 125 \cdot \#$ files	$80 + 72 \cdot \#$ files	n/a	$.56 + .3 \cdot \#$ files
Process Join	0	0	n/a	$14 + 10.7 \cdot \#$ files
Send Update	131	152	n/a	.6
Recv. Update	131	152	n/a	.8
Process Update	0	0	n/a	30
Packet Multiplex	0	0	0	$.01 \cdot \#$ open connections

Table 2.3: Costs of atomic actions

query and update. Each of these actions is composed of smaller "atomic" actions for which costs are given in Table 2.3. In this section, we discuss the atomic costs and how they are derived. In later chapters, we will discuss how we use atomic costs to then determine the costs of user-level actions.

There are two types of cost measured: bandwidth and processing power. In terms of bandwidth, the cost of an action is the number of bytes being transferred. We define the size of a message by the protocol in use, where applicable. For example, query messages in Gnutella include a 22-byte Gnutella header, a 2 byte field for flags and a null-terminated query string. Total size of a query message, including Ethernet and TCP/IP headers, is therefore 82 + query string length. Some values, such as the length of a query string, are not specified by the protocol, but are a function of the type of data being shared. These values, listed in Table 2.1, are described in Section 2.2. In addition, other values, such as the number of results returned for a query (# *results*) or the number of nodes that respond to a query (# *addr*), can be estimated using the query model developed in Section 2.3.

Note that there are a number of messages defined in Table 2.3 that have no counterpart in the actual protocol. For example, the Gnutella protocol has no Join or Update message standard defined. In such cases, we simply defined a message format that we felt was reasonable and fit in with the other messages in the protocol, and used that message format to determine bandwidth costs.

The processing costs of actions are given in coarse units, and were determined by measurements taken on a Pentium III 930 MHz processor running Linux kernel version 2.2. Processing costs will vary between machines and implementations, so the values seen in Table 2.3 are meant to be representative, rather than comprehensive. A unit is defined to be the cost of sending and receiving a Gnutella message with no payload, which was measured to be roughly 7200 cycles on the measurement machine. When displaying numbers and figures in the following chapters, we will convert these coarse units to cycles using this conversion ratio.

The packet multiplex cost is a per-message cost reflecting the growing operating system overhead of handling incoming and outgoing packets as the number of open connections increases. Please see Appendix A for a discussion of this cost and its derivation.

We note that in Table 2.3, processing costs are protocol-independent, while bandwidth costs are protocol-specific. Because we wanted to stay as true to each protocol as possible when performing comparison experiments, and because each protocol has a precise message protocol defined, we list bandwidth costs that exactly reflect these message protocols. However, the protocols do not specify the implementation of peers, therefore they do not specify exact processing costs. Instead, since nodes across each of the protocols should perform actions in a similar fashion, whether they be hybrid servers or peers in a pure network, we made the costs of each action the same across protocols.

Query Cost. The cost of processing a query consists of two main components: a startup cost that is constant across all queries, and the cost of reading the inverted lists (recall that in our model definition in Section 2.1, peers use inverted indices to process queries). Because it is difficult to determine how much of a list will be read to answer a query, we modeled the number of list elements accessed as a linear function of the number of results returned. The cost for answering a query is then $c_0 + c_1 \cdot \#\ \textit{results returned}$, where c_0 is the startup cost and c_1 is the cost associated with finding one result. By experimentally measuring the cost of a list read, and finding base costs of in-memory transaction overhead and record reads from [38], we determined that, roughly, the cost-per-result associated with processing a query is 4000 cycles (i.e., retrieving the full metadata from the database or file system and "packaging" the result into a message), and the cost-per-result associated with list access is 3778 cycles. Together, these costs turn into roughly 7778 cycles, or 1.1 coarse units of processing cost, per result.

Join/Leave Cost. When a node A joins the system, a cost is incurred on another node B if B maintains an index over A's collection. Four main costs are incurred on B: the startup cost/transaction overhead, the cost to read/process the incoming file metadata, the cost to write the file records to disk, and the cost to modify the inverted lists. Total cost is therefore $(a + b + c) \cdot \mathit{FilesPerUser} + d$, where a is the cost of reading/processing a file's metadata, b is the cost of writing a file record, c is the cost of modifying the inverted lists for one file's words and d is the startup cost. Using rule-of-thumb values for in-memory transaction overhead and record read/write from [38], we estimate values for a, b and d to be 2000, 5000 and 100000 instructions respectively (equivalently, .28, .69 and 14 coarse units, respectively).

If n files' metadata are uploaded to the server on login, then a total of $n \cdot \mathit{WordsPerMetadata}$ words are present in the uploaded metadata. Based on the indexes present on our hybrid server in the OpenNap experiments, we assume 60% of these words are distinct[4], and that all modifications that need to be made to a list are done in one batch modification. The result of the overlap is that only $0.6 \cdot n \cdot \mathit{WordsPerMetadata}$ lists must be modified in total. By experimentally determining the cost of list writes, we determined that modifying a list requires approximately 1.63 coarse units per list modification. The value for coefficient c is therefore $0.6 \cdot \mathit{WordsPerMetadata} \cdot 1.63 = 9.78$ units. Note that this cost is independent of how long the list is.

As in joining the system, leaving requires a startup cost. Furthermore, every list that was modified on node B during the join must now be modified during leave to remove the IDs of files belonging to the node A, and all records of these files must be removed from the database. Total cost for leaving is therefore the same as for joining.

Update Cost. From the perspective of the node maintaining an index, an insert action is similar to a join action, where the collection to be added consists of a single file. It involves the same equation as login: $(a + b + c) \cdot 1 + d$, where each coefficient has the same meaning. Each coefficient also has the same value, except for c. For join, we assumed that the words in the metadata had a significant overlap, so that only 60% of the total words were distinct. However, because there are so few words in a single file's metadata, we assume 0 overlap – that is, we assume every word is distinct. The value for c is therefore

[4]This is an approximation, since the amount of overlap in the collection of words would grow as the total grows.

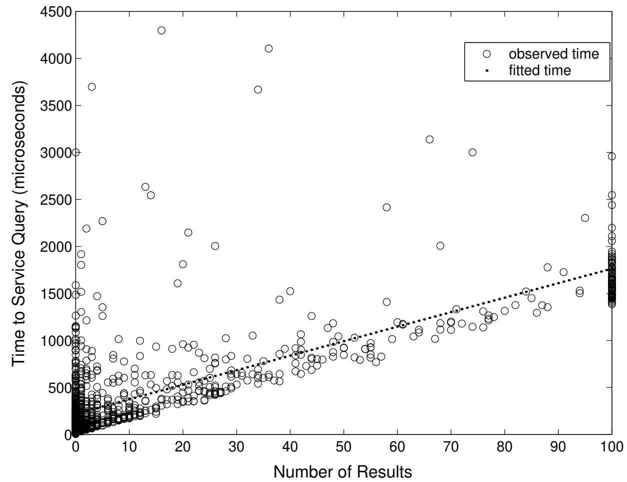

Figure 2.8: Time to Service a Query vs. Number of Results Returned

$WordsPerMetadata \cdot 1.63 = 16.3$. Total cost of an insertion is thus 31 coarse units.

Similarly, a delete action is similar to a leave action, where the collection to be removed consists of a single file. Therefore, we consider all updates to a single file to have the same cost.

Validation. The bandwidth component of the performance model does not require validation, as it follows the actual protocol specification. To validate the processing component, we ran measurement experiments over an actual hybrid server implementation, and compared the measurements to the costs predicted by our model.

For example, Figure 2.8 shows how well our model of processing costs for queries fits the observed data from our hybrid server. Figure 2.8 shows the plot of *time* to service a query for the given server, versus the number of results returned, as well as the analytic curve using the derived formula and CPU speed of the server. While the analytic curve is a rough approximation for a number of the datapoints, it does capture the general behavior of the majority of the datapoints. In some of our later evaluations (see Section 3.5), we study the effects of varying the cost-per-result coefficient (that is, the slope of the line in Figure 2.8) on system performance.

2.5 Metrics

We measure the performance of a file-sharing application based on two types of metrics: *load* and *quality of results*.

Load. When a message is propagated through the system, it passes through a number of nodes. Each of these nodes spends processing resources (i.e., cycles) on behalf of the message, whether it be to forward or process the message. Similarly, each node uses bandwidth resources on behalf of the message, as it sends and receives the message and any responses.

Load is defined as the amount of work an entity must do per unit of time, due to the costs described above. Load is measured along three resource types: *incoming bandwidth*, *outgoing bandwidth*, and *processing power*. Bandwidth is measured in bits per second (bps), processing power in cycles per second (Hz). Because load varies over time, we typically report values for average load over a period of time. We treat incoming and outgoing bandwidth as separate resources because their availability is often asymmetric: many types of connections (e.g., cable modem) allow greater downstream bandwidth than upstream. As a result, upstream bandwidth may become a bottleneck even if downstream bandwidth is abundant.

Some systems are efficient overall, while other systems may be less efficient, but have better load-balancing across nodes. Hence, we will look at both *individual* load, the load of a single node, as well as *aggregate* load, the sum of the loads of all nodes in the system. Note that with respect to aggregate load, there is no differentiation between incoming and outgoing bandwidth. Instead, we combine incoming and outgoing costs to form a single *aggregate bandwidth* measure.

Quality of Results. Although we seek to reduce the resources used to process the query, we must nevertheless ensure that our techniques do not degrade the user experience. We measure quality of results primarily by the *number of results* returned per query, and the *satisfaction rate* of the query – the percentage of queries that are satisfied (as defined earlier in this chapter). Other metrics for quality of results often include relevance of results and response time. Because relevance of results is subjective and application-specific, we generally do not use this metric. Also, although our simulations do not capture exact response times, relative response times can typically be deduced by other means, as will be seen in future chapters.

Note that not all chapters will use every metric, and some chapters may introduce new metrics that are specific to the techniques being evaluated.

2.6 Conclusion

Realistic application and user behavior models are necessary for the proper evaluation of a resource discovery mechanism. In this section, we modeled a file-sharing peer-to-peer application, and validated our models over live file-sharing applications. We found our models to be simple, and yet reflect realistic behavior. In subsequent chapters, we will use the models developed here as an evaluation tool for our algorithms and techniques.

Chapter 3

Server Architecture in Hybrid Networks

Our goal in this thesis is to study the scalability and functionality of P2P architectures. As there are many categories of possible architectures, in this chapter we focus on file-sharing *hybrid P2P systems*, as introduced in Chapter 1.

When Napster, a hybrid P2P system, was first introduced in 2000, it was described as the "fastest growing Internet application ever" [65]. Hundreds of thousands of users swarmed to Napster to share petabytes of digital music files, before the RIAA shut the company down in a landmark lawsuit. Despite their legal woes, Napster and other hybrid systems gave the technical community its first insight into the potential of large-scale P2P systems.

In a hybrid system, elements of both pure P2P and client/server systems coexist. For example, in Napster, a server node indexes files held by a set of users. (There can be multiple server nodes.) Users search for files at a server, and when they locate a file of interest, they download it directly from the peer computer that holds the file. In current and past applications, hybrid file-sharing systems have had better performance than pure systems because some tasks (like searching) can be done much more efficiently in a centralized manner.

Even though file-sharing hybrid P2P systems were hugely popular, there has been little scientific research done on them, and many questions remain unanswered. For instance, what is the best way to organize indexing servers? Should indexes be replicated at multiple servers? What types of queries do users typically submit in such systems? How should the system deal with users that are disconnected often (dial-in phone lines)? How will systems scale in the future when user interests and hardware capacity evolve? How do different query patterns affect performance of systems from different application domains?

In this chapter we attempt to answer some of these questions. In particular, the main contributions we make in this chapter are:

- We present (Section 3.2) several architectures for hybrid P2P servers, some of which are in use in existing P2P systems, and others that are new (though based on well-known distributed computing techniques).

- We present (Sections 3.3 and 3.4) extensions to the query and cost models presented in Chapter 2 that describe these different architectures.

- We provide (Section 3.5.1) a quantitative comparison of file-sharing hybrid P2P architectures, based on our query and performance models. Because both models are validated on a real music-sharing system, we begin experiments by focusing on systems in the music domain.

- We project (Section 3.5.2) future trends in user and system characteristics, analyzing how music-sharing systems will perform in response to future demands.

- We provide (Section 3.5.3) a comparison of strategies in domains other than music-sharing, showing how our models can be extended to a wide range of systems.

Many of the contributions in this chapter were first published in [94].

3.1 Related Work

Performance issues in hybrid file-sharing P2P systems have been compared to issues studied in information retrieval (IR) systems, since both systems provide a lookup service, and both use inverted lists. Much work has been done on optimizing inverted list and overall IR system performance (e.g., [59, 101]). However, while the IR domain has many ideas applicable to P2P systems, there are differences between the two types of systems such that many optimization techniques cannot be directly applied. For example, IR and P2P systems have large differences in update frequency. Large IR systems with static collections may choose to rebuild their index at infrequent intervals, but P2P systems experience updates every second, and must keep the data fresh. Common practices such as compression of indexes (e.g., [67]) make less sense in the dynamic P2P environment. While some work has been done in incremental updates for IR systems (e.g., [7, 89]), the techniques do not scale to the rate of change experienced by P2P systems.

Research in cooperative web caching (e.g., [25, 73]) has also led to architectures similar to the ones studied in this chapter. However, their models of performance are very different due to several key differences between their functionality and context and ours. First, most web caches search by key (i.e., URL), making hash-based cooperative web caches (e.g., [50]) more effective than hash-based P2P systems, which must allow multi-term keyword queries. Second, URL "queries" in web caches map to exactly one page, whereas queries in file-sharing systems return multiple results, and the number of results returned is an important metric of system effectiveness. In addition, modeling the probability of getting n hits in s servers is very different from modeling the probability of a single hit in s caches. Third, there is no concept of logins or downloads in web caching, which is a large part of P2P system performance. Fourth, bandwidth consumed by transferring a page from cache to client is an important consideration in cooperative web caching, but not in P2P systems, where files are transferred between peers. Finally, user behavior in the Web context is fairly well understood, whereas we will see how user behavior in P2P systems may vary widely depending on system purpose.

Several of the server architectures we present in the next section either exist in actual P2P systems, or draw inspiration from other domains. In particular, the "chained" architecture is based on the OpenNap [70] server implementation, and resembles the "local inverted list" strategy [76] used in IR systems, where documents are partitioned across nodes and indexed in subsets. The "full replication" architecture uses the NNTP [49] approach of fully replicating information across hosts, and a variation of the architecture is implemented by the Konspire P2P system [52]. The "hash" architecture resembles the "global inverted list" strategy [76] used in IR systems, where entire inverted lists are partitioned lexicographically across nodes. Finally, the "unchained" architecture is derived from the architecture used by Napster [62].

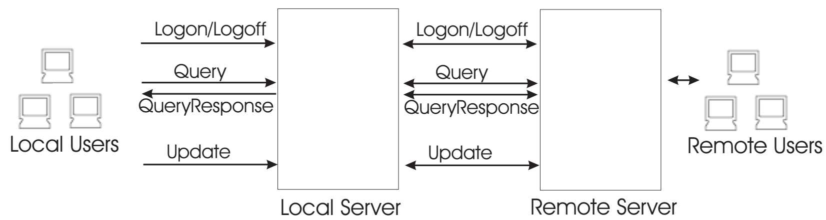

Figure 3.1: System Components and Messages

3.2 Server Architecture

We begin by describing how the basic concepts in a file-sharing system (presented in Section 2.1) pertain to a hybrid architecture, based on the OpenNap [70] implementation of a file-sharing service.[1] We then describe each of the architectures in general terms. Figure 3.1 shows the components in the basic hybrid P2P system and how they interact.

General Concepts. As discussed in Section 2.1, there are three basic actions supported by the system: join/leave, query and update.

To *join* and *leave* the system, a user must *login* to or *logoff* from a hybrid server. To login, a peer connects to a particular hybrid server, and uploads metadata describing the user's collection. The server maintains the *connection information* for all peers connected to it, such as client IP address and line speed.[2] By logging on at a server, the user is now able to send queries to its server, and is allowing other users to download her files. A hybrid system may contain multiple servers, but a user is logged in and connected to only one server, its *local server*. To that user, all other servers are considered *remote servers*. From the perspective of one server, users logged on to it directly are its *local users*. Depending on the architecture, servers may index the collections of both local and remote users.

When a server receives a *query*, it searches for matches in its index. The server sets a maximum number of results to return for any query. As defined earlier, a query is said to be satisfied if the maximum number of results are returned. As described in Section 2.1, a query is processed by retrieving the inverted lists for all its words, and intersecting the lists to obtain the identifiers of the matching documents. Clearly, other query and indexing schemes are possible (e.g., relational queries), and we discuss how to evaluate these schemes

[1] We are not following the exact protocol used by OpenNap, but rather use a simplified set of actions that represent the bulk of the activity in the system.

[2] Often, clients use dial-in connections, so their IP address can vary from connection to connection.

Name	Description	Default Value
FilesPerUser	Average files shared per user	168 files
RecordSize	Average size of result record	50 B
MetadataSize	Average size of metadata for a single file	50 B
WordsPerMetadata	Average number of words used to describe a file in its metadata	10
WordsPerQuery	Average number of words used in a query	2.4
QueriesPerUserSec	Rate of queries per user per second	$8.33 \cdot 10^{-4}$
QueryJoinRatio	Ratio of queries to joins	0.45
QueryDownloadRatio	Ratio of queries to downloads	0.5
FracChange	Fraction of user's collection changed offline	.1
ActiveFrac	Percent of the total user population that is active at any given time	.05
λ_f	Inverse frequency skew (see Section 2.3)	100
r	Query skew to occurrence skew ratio (see Section 2.3)	4

Table 3.1: User-Defined Parameters

Parameter Name	Default Value	Description
LANBandwidth	80 Mb/s	Bandwidth of LAN connection in Mb/s
WANBandwidth	8 Mb/s	Bandwidth of WAN connection in Mb/s
CPUSpeed	800 MHz	Speed of processor in MHz
NumServers	5	Number of servers in the system.
MaxResults	100	Maximum number of results returned for a query
User-Server Network	WAN	The type of network connection between users and servers
Inter-Server Network	LAN	The type of network connection between servers

Table 3.2: System-Defined Parameters

with our models in Section 3.5.3.

An *update* to a user's collection occurs after a successful download, or after a file is added through some other means. After an update, the client notifies the local server of the new addition to the collection. The local server will add this information to its index. The local server is also notified when local files are deleted. Depending on architecture, remote servers may also be notified of the addition/deletion of local files.

When a user logs off, the local server updates the index to indicate that the user's files are no longer available. Again, remote servers may have to update their indices as well, depending on architecture. The options for handling logoffs are discussed shortly.

Most hybrid file-sharing P2P systems offer additional services to users other than just file sharing, such as chat rooms and hot lists. These services are important in building community and keeping users attached to the main service. However, for our study we do not consider the effects of these activities on the system, as measurement experiments (described in Chapter 2) show that they do not significantly contribute to the workload.

A number of parameters may be used to describe the behavior of users and system

Parameter Name	Description
ExServ	Expected number of servers needed to satisfy a query
ExTotalResults	Expected number of results returned by all servers
ExLocalResults	Expected number of results returned by the local server
ExRemoteResults	Expected number of results returned by remote servers
UsersPerServer	Number of users logged on to each server

Table 3.3: Derived Parameters

characteristics. We show all parameters and their base values in Tables 3.1, 3.2, and 3.3, even though many of the parameters and their values will be described in later sections or were described earlier in Chapter 2. Parameters are divided into *user-dependent parameters* (Table 3.1) – those parameters that describe characteristics of user behavior, *system parameters* (Table 3.2) – those parameters that determine available system resources, and *derived parameters* (Table 3.3) – those parameters that are derived from other user and system parameters. The last derived parameter, *UsersPerServer*, is the value we want to maximize for each server. Our performance model calculates the maximum users per server supportable by the system, given all other parameters. User behavior parameters have already been introduced in Section 2.2, but we repeat them in Table 3.1 for reader convenience.

Batch and Incremental Logins. In current hybrid P2P systems, when a user logs on, metadata on her entire collection is uploaded to the server and added to the index. Similarly, when she logs off, all of her collection information is removed from the index. At any given time, only the libraries of connected, or active, users are in the index. We call this approach the *batch* policy for logging in. While this policy allows the index to remain as small as possible and thereby increases query efficiency, it also generates expensive update activity during login and logoff.

An alternative is an *incremental* policy where user files are kept in the index at all times. When a user logs on, only files that were added or removed since the last logoff are reported. If few user files change when a user is offline, then incremental logins save substantial effort during login and logoff. (The parameter *FracChange* tells us what fraction of files change when a user is offline.) However, keeping file metadata of *all* users requires filtering query results so that files belonging to inactive users are not returned. This requirement creates a performance penalty on queries. Also notice that a user must always reconnect to the same server, at least with the chained servers we have described so far. This restriction may be a disadvantage in some applications.

	Chained	**Full Replication**	**Hash**	**Unchained**
Login	Index collection metadata.	Index collection metadata, forward metadata to all remote servers.	Hash metadata words, forward metadata to appropriate servers.	Index collection metadata.
Query	Search local index. While not satisfied, forward query to remote servers.	Search local index.	Hash query terms, retrieve necessary inv. lists, and merge.	Search local index.
Down-load	Update local index.	Update local index, forward metadata to all remote servers.	Hash metadata words, forward metadata to appropriate servers.	Update local index.

Table 3.4: Architecture Summary

Chained Architecture. In this architecture, the servers form a linear chain that is used in answering queries. When a user first logs on, only the local server adds collection metadata to its index; remote servers are unaffected. When a user submits a query, the local server attempts to satisfy the query alone. However, if the local server cannot find the maximum number of results, it will forward the query to a remote server along the chain. The remote server will return any results it finds back to the first server, which will then forward the results to the user. The local server continues to send the query out to the remaining remote servers in the chain until the maximum number of results have been found, or until all servers have received and serviced the query. In this architecture, logins and downloads are fast and scalable because they affect only the local server of a user. However, queries are potentially expensive if many servers in the chain are needed to satisfy the query.

Full Replication Architecture. Forwarding queries to other servers can be expensive: each new server must process the query, results must be sent to the originating server, and the results must be merged. The full replication architecture avoids these costs by maintaining on each server a complete index of all user files, so all queries can be answered at a single server. Even with incremental logins, users can now connect to any server. The drawback, however, is that all logins must now be sent to every server, so that every server can maintain their copy of the index (and of the user connection information). Depending on the frequency ratio of logins to queries, and on the size of the index, this may or may not be a good tradeoff. If servers are connected by a broadcast network, then login propagation can be more efficient.

Hash Architecture. In this scheme, metadata words are hashed to different servers, so that a given server holds the complete inverted list for a subset of all words. We assume words are hashed in such a way that the workload at each server is roughly equal. When a user submits a query, we assume it is directed to a server that contains the list of at least one of the keywords. That server then asks the remaining servers involved for the rest of the inverted lists. When lists arrive, they are merged in the usual fashion to produce results. When a client logs on, metadata on its files (and connection information) must be sent to the servers containing lists for the words in the metadata. Each server then extracts and indexes the relevant words.

This scheme has some of the same benefits of full replication, because a limited number of servers are involved in each query, and remote results need not be sent between servers. Furthermore, only a limited number of servers must add file metadata on each login, so it is less expensive than full replication for logins. The main drawback of the hash scheme is the bandwidth necessary to send lists between servers. There are several ways to make this list exchange more efficient [88]; we describe one such technique in Section 3.4.

Unchained Architecture. The "unchained" architecture simply consists of a set of independent servers that do not communicate with each other. A user who logs on to one server can only see the files of other users at the same local server. This architecture, used by Napster, has a clear disadvantage of not allowing users to see all other users in the system. However, it also has a clear advantage of scaling linearly with the number of servers in the system. Though we cannot fairly compare this architecture with the rest (it provides partial search functionality), we still study its characteristics as a "best case scenario" for performance.

3.3 Query Model

Using the basic query model developed in Section 2.3, we must compute the "derived parameters" described in Table 3.3: *ExServ*, *ExTotalResults*, *ExLocalResults* and *ExRemoteResults*.

First, let us consider computing derived parameters under the chained architecture. The expected number of servers that must process a query to satisfy it, *ExServ*, is simply $K(n, R, S)$, where R is defined as the maximum number of results returned by the system, S

is the number of servers in the chain, and n is the number of files indexed by each server. In other words, using the system parameters defined in Table 3.2, *ExServ* $= K($*UsersPerServer*$\cdot$ *FilesPerUser*, *MaxResults*, *NumServers*$)$. The expected number of total, local and remote results returned is simply $M(n, R)$, where R is again the maximum number of results returned by the system, and n is replaced by the number of files indexed by all servers, the local server, and the remote servers, respectively. In other words, for local results, $n = n_l =$ *UsersPerServer* $\cdot$ *FilesPerUser*; for remote results, $n = n_r =$ *UsersPerServer* $\cdot$ *FilesPerUser* $\cdot$ (*NumServers* $- 1$); for total results, $n = n_l + n_r$.

For the full replication architecture, *ExServ* is trivially 1, since every server contains a full replica of the index at all other servers. Because *ExServ* is 1, all results are local, and none are remote. Hence *ExRemoteResults* $= 0$, and *ExLocalResults* $=$ *ExTotalResults* $= M(n, R)$, where $n =$ *NumServers*$\cdot$ *UsersPerServer* $\cdot$ *FilesPerUser* is the number of files in the entire system.

For the unchained architecture, *ExServ* is trivially 1, since queries are not sent to remote servers. *ExRemoteResults* $= 0$ and *ExLocalResults* $=$ *ExTotalResults* $= M(n, R)$, where $n =$*UsersPerServer* $\cdot$ *FilesPerUser* is the number of files at a *single* server.

Finally, for the hash architecture, again *ExRemoteResults* $= 0$ and *ExLocalResults* $=$ $M($*NumServers* $\cdot$ *UsersPerServer* $\cdot$ *FilesPerUser*$, R)$, since all the results are found at the server that the client is connected to. *ExServ* is more difficult to calculate, however. The problem again takes the form of the definition of $K(n, R, S)$ (Equation 2.13), but instead of using the probability $P(s, n, R)$ (the probability that exactly s servers are needed to return at least R results, where n is the number of files indexed by each server), we want to use a different probability $P_2(s, n, R)$ that exactly s servers are "involved" in answering the query. A server is "involved" in a query if it contains at least one of the inverted lists needed to answer the query. We assume that on average, *WordsPerQuery* lists are needed to answer a query; however, it is possible for more than one list to exist at the same server. The problem of finding the probability of b lists residing at exactly i servers can be restated as the probability of b balls being assigned to exactly i bins, where each ball is equally likely to be assigned to any of the k total bins. Because the average number of words per query is typically quite small, we are able to explicitly compute the necessary values, and interpolate the values to get a close approximation in the case of fractional *WordsPerQuery*. For example, say *WordsPerQuery* $= 3$ and *NumServers* $= 5$. Then the probability of all lists residing at one, or at 3 distinct servers is:

$$\text{At 1 server: } \binom{5}{1}\left(\frac{1}{5}\right)^3 = .04 \qquad \text{At 3 distinct servers: } \binom{5}{3} 3! \left(\frac{1}{5}\right)^3 = .48 \quad (3.1)$$

Finally, the probability of all lists residing at 2 distinct servers is 1 minus the probabilities of 1 and 3 distinct servers, which is .48. The expected number of servers involved in a query with 3 words over a set of 5 servers is therefore 2.44. Please refer to Appendix B.1 for a complete solution.

3.4 Performance Model

In this section, we describe the performance model used to evaluate the architectures. We begin by defining our system environment and the resources our model will consider. We then describe how we use the basic cost model from Section 2.4 to calculate the cost of actions for different architectures. Finally, we describe how we put the parameters and costs together to determine the performance of the system.

In terms of metrics, our focus will be on the load on the hybrid servers, rather than aggregate load of the entire system, since client load is insignificant in a hybrid system. In addition, because server resources are constrained, we do not simply report the average cost of a query in terms of bandwidth and processing power, as described in Section 2.5. Instead, we will combine costs and constraints to determine the maximum number of users a system can support.

3.4.1 System Environment and Resources

In our model, we assume the system consists of *NumServers* identical server machines. The machines are connected to each other via a broadcast local area network (LAN), for example, if they all belong to the same organization, or through a point-to-point wide area network (WAN). Similarly users may be connected to servers via LAN or WAN. Our system can model any of the four combinations; let us assume for the examples in the remainder of the section that servers are connected to each other via LAN, and to users via WAN.

We calculate the cost of actions in terms of three system resources: CPU cycles, inter-server communication bandwidth and user-server communication bandwidth. These costs are calculated as described in the next section, using the atomic action costs provided in

Chapter 2. If users and servers are all connected via the same network (e.g., the same LAN), then the last two resources are combined into one. For the time being, we do not take I/O costs into consideration, but assume that memory is cheap and all indexes may be kept in memory. Memory is a relatively flexible resource, and depends largely on the decisions of the system administrator; hence, we did not want to impose a limit on it. Later in Section 3.5, we will discuss the memory requirements of each architecture, and point out the tradeoffs one must make if a limit on memory is imposed.

Table 3.2 lists the default system parameter values used in our performance analysis. The bandwidth values are based on commercial enterprise level standards, while the CPU represents current high-end commodity hardware. *NumServers* and *MaxResults* were taken from the OpenNap chain settings.

3.4.2 CPU Consumption

The CPU cost of actions vary depending on architecture. In this section, we enumerate these actions and describe how the costs change under different architectures and login policy.

Login. In the chained and unchained architectures, the cost follows the formula for `Process Join`, given in Table 2.3 in Section 2.4. In the full replication architecture, login/logoff follows the same procedure as in chained, except that the procedure must be executed at every server. Hence, total instructions (across all servers) for login/logoff is *NumServers* times the cost for the chained architecture. Finally, in the hash architecture, the login/logoff process is also similar, except that every server might receive only a portion of the file metadata. With the given parameters, we have estimated in Appendix B.1 the number of servers, s, at which a file will be replicated. Total login cost for hash is thus s times the cost for the chained architecture.

For a batch login, the "*# files*" parameter for the formula in Table 2.3 is *FilesPerUser*. For an incremental login. this parameter is $\mathit{FilesPerUser} \cdot \mathit{FracChange}$.

Query. The cost of a query in the unchained and full-replication architectures follow the formula for `Process Query` given in Table 2.3, where the "*# results*" is set to *ExTotalResults*. In the chained and hash architectures, cost is similar to unchained, except that we multiply the startup cost, 14 units, by *ExServ*, the number of servers that will process the query.

Update. The cost of an update in the chained and unchained architectures follow the formula for `Process Update` given in Table 2.3. As with login, the cost of an update is multiplied by *NumServers* in the full replication architecture, since every server must process the update. Similarly, the cost of update is multiplied by s in the hash architecture, where s is the number of servers to which the metadata words hash.

3.4.3 Network consumption

Again, network cost of actions vary depending on architecture.

Login. When a client logs in to a server in batch mode, a single Login message is sent from client to server, as well as one `AddFile` message for every file to be added to the server's index,[3] The user-server bandwidth necessary for a login therefore follows the formula for `Send Join` in Table 2.3 in Section 2.4, for all architectures using batch logins, using *FilesPerUser* as the value for the "*# files*" parameter.

With incremental logins, again, a single Login message is sent from client to server, as well as one `AddFile` message for every file to be added, and one `RemoveFile` message for every file to be removed. We predict the number of files *changed* since the last login as $FilesPerUser \cdot FracChange$. We assume that half the changes are file deletions, and half the changes are file additions, since on average, this must be the case if the average number of files per user remains constant. Thus, user-sever bandwidth cost of an incremental login is $42 + 125 \cdot \#$ files added $+\ 76 \cdot \#$ files removed, where 125 is the size of an AddFile message and 76 is the size of a RemoveFile message, in bytes.

For inter-server communication, the bandwidth required for a login depends not only on login policy, but on architecture as well. For the chained and unchained architectures, logins are not replicated across servers, hence 0 bytes/login are required. For the full replication architecture, login messages are sent to every server. If servers are connected via broadcast LAN, then the messages need only be replicated once, meaning inter-server cost is the same as user-server cost. If servers are connected via point-to-point WAN, then communication cost increases by a factor of $NumServers - 1$. Similarly, in the hash architecture, if servers

[3] Yes, a separate message is sent to the server for each file in the user's collection. This is inefficient, but it is the way the Napster protocol operates.

are connected via LAN, then messages need only be replicated once, whereas if servers are connected via WAN, then the messages must be replicated $s-1$ times, where s is the number of servers a file's metadata hashes to (Appendix B.1).

Query. For user-server communication, query cost is as listed in Table 2.3 for sending and receiving queries and responses, using *ExTotalResults* as the value for the "*# files*" parameter. This cost is the same across all architectures and login policies.

For inter-server communication, in the full replication and unchained architectures, servers do not send queries to each other, hence the cost is zero.

The chained architecture sends the Query message to as many servers as are necessary to satisfy the query, and a QueryResponse message is sent from remote server to local server for every remote result. The inter-server cost formula for sending a query is thus $ExServ-1$ times the cost of user-server communication in a WAN, or 1 times the cost in a LAN. The cost formula for transferring responses is according to the `Send Response` formula in Table 2.3, but using *ExRemoteResults* as the value for the "*# files*" parameter.

In the hash architecture, the Query message must be sent out to the s servers that contain the appropriate inverted lists, as calculated in Section 3.3. The inter-server cost for sending a query in a WAN is thus $s-1$ times the cost of user-server communication, or 1 times the cost if using a LAN, and if $s > 1$. In addition, the inverted lists at the remote servers need to be transferred over to the local server.

Update. On update, a single Update message must be sent from user to server. In terms of inter-server communication, in the chained and unchained architectures, servers do not notify each other of updates; therefore the cost is zero. In the full replication architecture, every server must be notified of every update. Hence if servers are connected via WAN, the update message is resent $NumServers-1$ times; otherwise on a LAN, it is resent 1 time. Similarly, in the hash architecture, the file metadata must be replicated as many servers as it would during login.

3.4.4 Memory Requirements

In the batch chained architecture, for every active user, the local server must keep a user record, estimated at 64 bytes given the implementation of the OpenNap hybrid server. For every file in the user's collection, the server must keep a file record, similarly estimated at

128 bytes. Furthermore, every file has $WordsPerMetadata = 10$ words in the metadata, and for every word, an ID for that file is stored in the appropriate inverted list, where an ID is estimated at 8 bytes. Hence, total memory usage per active user is: $64 + 128 \cdot FilesPerUser + 8 \cdot WordsPerMetadata \cdot FilesPerUser$

The unchained architecture implementing the batch policy requires the same amount of memory as the chained. In the full replication architecture, every user's information is replicated at each server, so that total memory usage is multiplied by *NumServers*. In the hash architecture, every user record is replicated at each server. In addition, a file is replicated at roughly s servers, where s is calculated in Appendix B.1. However, inverted lists are *not* replicated across servers, so for every word in the file metadata, only one ID for that file is stored in the appropriate inverted list, as in chained. Total memory usage per active user in hash is therefore: $64 \cdot NumServers + 128 \cdot s \cdot FilesPerUser + 8 \cdot WordsPerMetadata \cdot FilesPerUser$.

For all architectures, if the incremental policy is implemented rather than the batch policy, then every server holds $\frac{1}{ActiveFrac}$ times as many users as are held with the batch policy. Hence, memory usage is $\frac{1}{ActiveFrac}$ times greater per active user.

3.4.5 Overall Performance

After deriving formulas to describe the cost of each action, we can put everything together to determine how many users are supportable by the system. Our end goal is to determine a maximum value for *UsersPerServer*. To do this, we first calculate the maximum number of users supportable by each separate resource, assuming infinite resources of the other two types. Then, to find the overall *UsersPerServer*, we take the minimum across the three resources. In other words, finding the maximum value for *UsersPerServer* is a linear programming problem, in which the constraints are that individual resources (CPU, user-server communication, inter-server communication) are not exceeded.

3.5 Experiments

In this section, we present the results of our performance studies, discussing the behavior of the architectures and login policies as certain key parameters are varied, and highlighting the tradeoffs between architectures in different scenarios. We begin by studying music-sharing systems today (Section 3.5.1). Next, we consider how such systems might perform

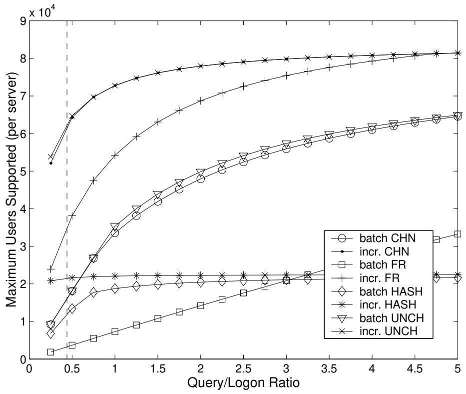

Figure 3.2: Overall performance of strategies as *QueryJoinRatio* is varied

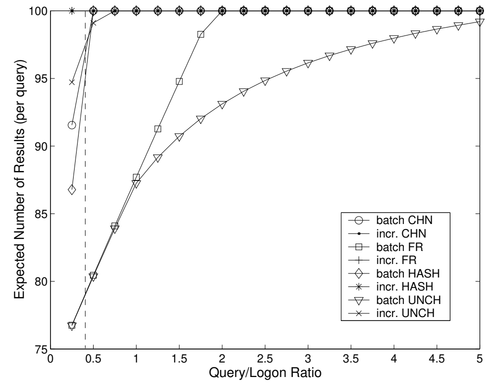

Figure 3.3: Expected number of results as *QueryJoinRatio* is varied

in the future when user and system characteristics have changed (Section 3.5.2). Finally, in Section 3.5.3 we switch focus to other domains and evaluate the performance of systems that may have very different query and indexing characteristics from music-sharing applications.

Throughout the performance comparisons, the metric of performance is the maximum number of users each server can support. Hence, we concern ourselves with throughput, and not response time. Unless otherwise specified, default values for parameters (see Tables 3.1, 3.2, and 3.3) were used during evaluation. For brevity, we will refer to the chained architecture as CHN, the full replication architecture as FR, the hash architecture as HASH, and the unchained architecture as UNCH. Because each architecture can be implemented using one of two login policies, we refer to the combination of a particular architecture and policy as a *strategy*. For example, "batch CHN" is the strategy where the chained architecture is implemented using the batch login policy. There are a total of 8 strategies.

3.5.1 Music-Sharing Today

We begin by evaluating performance of systems with behavior similar to that of music-sharing systems today (e.g., Napster, OpenNap), described by default user and system parameter values listed in Tables 3.1 and 3.2. Figure 3.2 shows the overall performance of the strategies over various values of *QueryJoinRatio*. For example, at $QueryJoinRatio = 1$, incremental FR can support 54203 users per server, whereas batch FR can only support 7281 users per server. The dashed line represents the experimentally derived value of *QueryJoinRatio* for OpenNap. Figure 3.3 shows the expected number of results per query for each of the strategies shown in Figure 3.2, assuming that *MaxResults* = 100. As *QueryJoinRatio* increases, the number of logins per second decreases, meaning more users can be supported by the system, thereby making more files available to be searched. As a result, when *QueryJoinRatio* increases, the expected number of results increases as well.

As seen in Figure 3.2, the incremental CHN and UNCH strategies have the best performance. Both strategies are bound by user-server communication over WAN. The FR strategies are bound by CPU due to expensive logins. However, as *QueryJoinRatio* increases, logins become less of a factor in overall performance, and we see how incremental FR performance improves until *QueryJoinRatio* = 5, where user-server communication becomes the bottleneck and the strategy has equal performance to incremental CHN and UNCH. Finally, HASH is bound by inter-server communication over LAN because of the large lists that must be transferred during queries. In many of the scenarios we study in later figures, we will see that as a general rule, the FR bottleneck is CPU when *QueryJoinRatio* is low, and the HASH bottleneck is inter-server communication.

We can also see in Figure 3.2 that user-server communication for incremental strategies, as seen in the performance of incremental CHN, is better than batch user-server communication performance, as seen in the performance of batch CHN. In fact, incremental user-server communication performance is always better than batch performance, because while queries consume the same bandwidth in both policies, logins use significantly less bandwidth in incremental than in batch. Furthermore, since all architectures adhere to the same message protocol for user-server communication, all architectures have roughly the same user-server communication performance when operating under the same login policy.

Now, let us consider only the *batch* login policy, which is the policy we believe to be closest to Napster. The top two batch strategies, CHN and UNCH, have roughly the same performance. However, batch UNCH returns substantially fewer results per query than batch CHN. Figure 3.3 shows the expected number of results per query, assuming that *MaxResults* = 100. As *QueryJoinRatio* increases, the number of logins per second decreases. As a result, more users can be supported by the system, meaning more files are available to be searched, and more results are returned. Most strategies reach the maximum quickly, but batch UNCH has a substantially lower expected number of results, and "catches up" as *QueryJoinRatio* increases. By the time expected number of results reaches *MaxResults*, batch UNCH has the same exact performance as batch CHN. Hence, batch UNCH only has better performance than batch CHN when it returns fewer results; furthermore, the performance difference is very small, while the expected number of results difference can be quite large. If the current parameters do indeed describe Napster's systems, then the benefits of UNCH in capacity may not outweigh the disadvantage of fewer results, and batch CHN would be a better strategy than the current batch UNCH strategy.

In summary, from this experiment we can make several important conclusions:

- Incremental strategies outperform their batch counterparts. In particular, incremental CHN and UNCH have the best performance and are recommended in this scenario. Currently, the incremental policy is not used by music-sharing systems, but it should clearly be considered (see end of section for memory requirement considerations).

- In a network-bound system, incremental strategies will always outperform batch because while queries consume the same bandwidth in both policies, logins use significantly less bandwidth in incremental than in batch.

- Batch UNCH is the strategy that most closely describes Napster's architecture. As seen in the figures, surprisingly, adding chaining to the servers (switching from UNCH to CHN) does not affect performance by much, but returns significantly more results. Assuming the cost of maintaining a LAN between the servers is acceptable, batch CHN is clearly recommended over batch UNCH.

- Most policies are very sensitive to *QueryJoinRatio* near our measured value of 0.45. Small changes in *QueryJoinRatio* can significantly increase or reduce the maximum number of users supported by the system, thereby making capacity planning difficult. This sensitivity is especially important to consider if large increases in *QueryJoinRatio* are expected in the future when user network connections become more stable (see Section 3.5.2).

- As a general rule, FR is CPU-bound due to expensive logins when *QueryJoinRatio* is small, HASH is bound by inter-server communication because of the large lists that must be transferred during queries, and CHN and UNCH are bound by user-server communication.

From this point on, we will no longer include the unchained architecture in our comparisons because the tradeoff is always the same: better performance, but fewer results per query.

Batch versus Incremental CPU Performance. In Figure 3.2, we saw that incremental strategies always have better user-server communication performance than batch strategies. Now we would like to see how they compare in terms of CPU performance. From the formulas in Section 3.4, the incremental policy clearly has better login performance, especially as *FracChange* decreases. However, incremental also has a much worse query performance because the server must filter out inactive file information. Therefore, which strategy is better should depend on the ratio of queries to logins that the server must handle. In Figure 3.4, we see the CPU performance (that is, maximum users supported by the CPU, assuming network communication is not the bottleneck) of each strategy as *QueryJoinRatio* varies in value. As expected:

- In a CPU bound system, incremental strategies only outperform batch when *QueryJoinRatio* is small, because a small ratio of queries to logins is in their favor. Note that in the previous experiment, the system was partially network bound, so this rule does not apply to that scenario.

- Batch strategies are recommended over incremental strategies in a CPU bound system where *QueryJoinRatio* is large. As seen in the figure, the CPU performance of all batch strategies improves as *QueryJoinRatio* increases, until it surpasses the performance of the incremental strategies.

Note that the balance between incremental and batch depends on *FracChange* and *ActiveFrac* as well. If *FracChange* is small and/or *ActiveFrac* is large, then the balance favors the incremental strategy. The effect of these two parameters is discussed in further detail in Appendix B.2.

Memory Requirements. Thus far, we have evaluated the strategies assuming that there was enough memory to hold whatever indexes a server needed. We will now take a closer look at the memory requirements of each strategy.

Figure 3.5 shows the memory requirement in bytes of the various strategies, as a function of the number of users. Here, we assume $ActiveFrac = .1$, to keep all architectures within roughly the same scale. Clearly, the batch strategies are far more scalable than the incremental strategies. For example, when there are 10000 users in the system, batch CHN requires .35 GB of memory, while incremental CHN requires 10 times that amount. Also, CHN requires the least amount of memory, while FR requires the most. However, it is

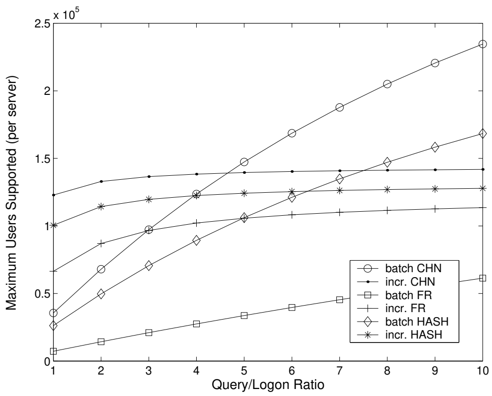

Figure 3.4: CPU performance of login policies as *QueryJoinRatio* is varied

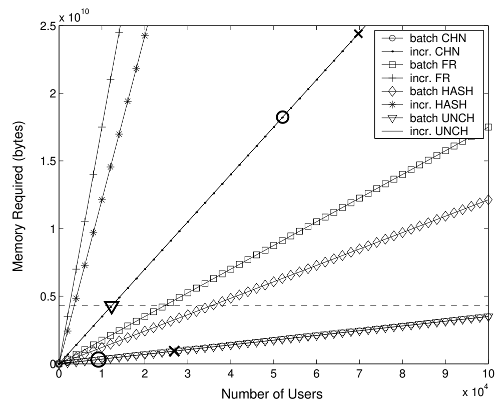

Figure 3.5: Memory requirements

important to note that memory requirement is a function of several parameters, most importantly *NumServers* and *ActiveFrac*. As *NumServers* decreases, FR requires proportionally less memory. On the flip side, as *NumServers* increases, FR also requires proportionally more memory. Likewise, incremental strategies require $1/ActiveFrac$ as much memory as batch. As connections become more stable and *ActiveFrac* increases, memory required by incremental strategies will decrease inverse proportionally, until it is much more comparable to batch memory requirements than it currently is. Furthermore, as memory becomes cheaper and 64-bit architectures becomes widespread, memory limitations will become much less of an issue than it is now.

Today, it is likely that system administrators will limit the memory available on each server. By imposing a limit, several new tradeoffs come into play. For example, suppose a 4GB memory limit is imposed on each server, shown by the dashed line in Figure 3.5. Now, consider a Napster scenario where $r = 4$, $\lambda_f = 100$ and $ActiveFrac = .1$. Say we determine that $QueryJoinRatio = .75$. Our model predicts that the maximum number of users supported by batch CHN is 26828, and by incremental CHN is 69708. The memory required by these two strategies is shown in Figure 3.5 by the large 'x' marks. While incremental CHN can support over twice as many users as batch CHN, it also requires very large amounts of memory – far beyond the 4GB limit. If we use the incremental CHN strategy with the 4GB limit, then our system can only supported 12268 users per server, shown as a large triangle in Figure 3.5, which is fewer than the users supported by batch CHN. Hence, batch CHN is the preferred architecture for this scenario.

However, let us now suppose *QueryJoinRatio* is .25. Then, the maximum number of users supported by batch CHN is 9190, and by incremental CHN is 52088. The memory required by these two strategies is shown in Figure 3.5 by the large 'o' marks. Again, the amount of memory required by incremental CHN is far too large for our limit. However, looking at incremental CHN performance at the 4 GB limit, we find that the 12268 users supported by incremental CHN is greater than the 9190 users supported by batch CHN. Hence, incremental CHN is still the better choice, because within the limit, it still has better performance than batch CHN.

3.5.2 Music-Sharing in the Future

In the future, a number of user and system characteristics can change, thereby affecting the performance of music-sharing systems. Here we consider three types of changes: change in network stability, change in user query behavior, and change in system hardware capacity.

Changes in Network Stability. As mentioned in Section 2.2, the experimentally observed value for *QueryJoinRatio* is surprisingly low, and this is most likely due to the fact that most users today have unstable network connections through dialup modems. In the future, we would expect more users to have stable, high bandwidth connections from home through DSL or cable modem. How would system performance change given this trend in network stability? First, if connections are stable and users can remain online for longer periods of time, then we would expect *QueryJoinRatio*, the number of queries a user submits per session, to rise from its current value of 0.45. *QueryJoinRatio* $= 0.45$ means that the average user submits 1 query every other time s/he logs on, which is very low. We would expect that in the future, *QueryJoinRatio* is at least an order of magnitude larger, so as a future estimate we will use *QueryJoinRatio* $= 10$.

Second, if users remain online for longer periods of time, then at any given moment, a higher fraction of total users are logged in. The current value for *ActiveFrac* $= 0.05$, meaning 1 out of 20 users are logged in at any given time. Though *ActiveFrac* will increase, it probably will not increase as much as *QueryJoinRatio*, so as a future estimate, we will use *ActiveFrac* $= 0.25$, meaning 1 out of 4 users are logged in at any given time.

Figure 3.6 shows the performance of the strategies given the default parameter values listed in Tables 3.1 and 3.2, but with *QueryJoinRatio* $= 10$ and *ActiveFrac* $= 0.25$. We show performance as a function of the number of servers in the system, in order to compare

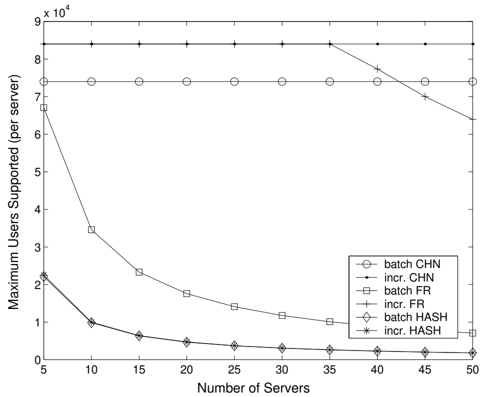

Figure 3.6: Scalability in stable networks

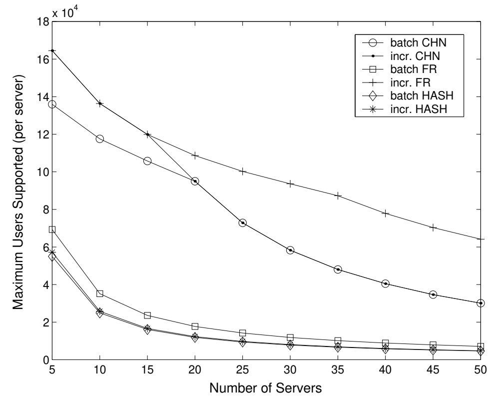

Figure 3.7: Scalability with diverse interests

the scalability of the strategies. Note that the vertical axis shows the maximum number of users *per server*. A scalable strategy should exhibit a fairly flat curve in this figure, because as more servers are added to the system, the capacity at each server remains near-constant and the system will have best-case linear scalability. The lessons in scalability we gather from this experiment are:

- Incremental CHN continues to be the best performing strategy, and one of the most scalable as well. CHN's scalability is due to the fact that most operations involve only the local server. Logins and updates are handled at the local server only, and because *ExServ* is very low with $\lambda = 100$ and $r = 4$, queries are mostly handled at the local server as well. In the next experiment, we see how CHN scales poorly when *ExServ* is high.

- FR is not scalable because the cost of updates and logins increase in proportion to the number of servers in the system. Since the incremental policy has better login performance than batch, incremental FR has good performance until the number of servers becomes large (˜35). However, batch FR is always unscalable because of the CPU cost of logins. If CPU speeds increase such that CPU is no longer the bottleneck, batch and incremental FR would perform and scale as well as batch and incremental CHN, respectively.

- HASH is not scalable in this scenario because all operations become more expensive as the number of servers increase, in terms of inter-server communication. However, we show next that if network bandwidth increases such that it is no longer the bottleneck, HASH can have the best performance and scalability. In addition, we show that when the number of words per query is low, HASH is by far the most scalable.

Changes in Query Model. Future changes in query model parameters will also affect the performance of strategies. For example, suppose user interests become more diverse. In this case, query popularity and selectivity are less skewed, meaning λ and r are larger. In addition, diverse interests translate into fewer expected results in the query model, meaning users will likely submit more queries before finding the files they wish to update. In this case, *QueryUpdateRatio*, the number of queries submitted before a single update, will increase as well.

Figure 3.7 shows us performance of strategies when user interests are diverse, that is, where $r = 16$, $\lambda = 1000$ and $QueryUpdateRatio = 2$. In Napster, $\lambda = 100$ and $r = 4$, meaning the $100th$ query had the mean selectivity, and the $400th$ query had the mean popularity. In this scenario, both distributions are more evenly distributed, so that only the $1000th$ query has the mean selectivity, and the $16000th$ query has the mean popularity. With OpenNap, $QueryUpdateRatio = 0.5$, meaning a user updated 2 files for every query submitted. Now, with fewer expected results per query, we estimate that users will only update a file with every other query submitted, meaning $QueryUpdateRatio = 2$. In this figure, we also assume that network connections are stable as in the previous example, meaning $QueryJoinRatio = 10$ and $ActiveFrac = 0.25$.

We see in this scenario that incremental FR has the best performance. Although FR is still bound by CPU, now CHN is bound by inter-server communication, because diverse interests mean fewer results, and fewer results means higher *ExServ* for CHN. CHN does not scale well with high *ExServ*, because many query and query result messages must then be exchanged between servers. FR and HASH, however, are relatively unaffected by changes in r and λ, because *ExServ* in these strategies do not depend on these parameters.

Future changes to the query model are difficult to predict; however, from the above experiment and other experiments not shown, we observe several general trends:

- Changes in r have a much larger effect on system performance than changes in λ. Changes in λ are generally only noticeable when r remains the same.

- If r and/or λ decreases, then the distributions are more skewed (e.g., "narrow interests"), and *ExServ* remains small. As a result, CHN queries are still scalable, and

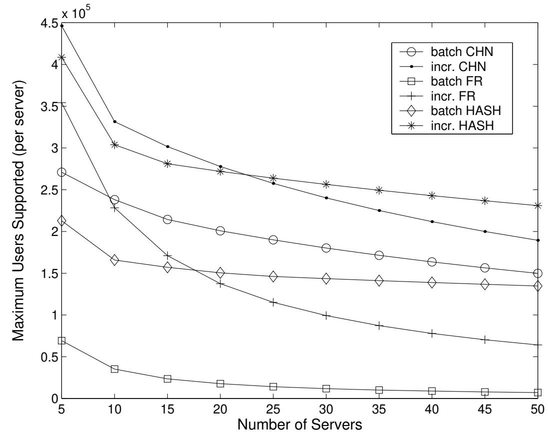

Figure 3.8: Scalability with unlimited bandwidth

relative performance of the strategies is unaffected by the changes. Incremental CHN continues to have the best performance and scalability.

- If r and/or λ increases, then the distributions are more evenly distributed (e.g., "diverse interests"), and *ExServ* increases. As a result, CHN queries are unscalable, and incremental FR has the best performance and scalability.

Changes in Hardware Capacity. In the future, we may also expect changes in hardware capacity. For example, suppose network bandwidth for both WAN and LAN increases such that network is no longer the bottleneck for any of the strategies. Figure 3.8 shows the same scenario illustrated by Figure 3.7, but where CPU is the limiting resource for all strategies. We see from this figure that HASH is now the most scalable of the strategies. After the system scales beyond 10 servers, both batch and incremental HASH curves become rather flat, and after 20 servers, incremental HASH has the best performance. HASH is scalable in regards to CPU because beyond a certain point, both login and query performance are almost unaffected by the number of servers in the system. For logins, every file is replicated at most *WordsPerMetadata* times, if every word in the metadata hashes to a different server. Since $WordsPerMetadata = 10$, after $NumServers = 10$, the degree of replication barely increases as the number of servers increase. Login notices are still sent to every server, but the size of a notice is very small compared to collection metadata. For queries, at most *WordsPerQuery* servers are involved in a query. Hence, beyond $NumServers = 2.4$, the number of servers involved in a query remains basically constant. The other strategies are not as scalable as HASH for reasons mentioned before. CHN is very scalable in terms of logins, but not queries if *ExServ* is high (e.g., if r is high). As the system scales up, more servers are involved in satisfying a query, so performance per server decreases. FR is very scalable in queries but not logins, so as the number of servers increases, logins take a proportionally larger slice of the CPU. In general, changes in hardware capacity will only improve the performance of those strategies that currently experience a bottleneck with that resource.

3.5.3 Beyond Napster: Systems in Other Domains

Thus far, we have considered systems that are similar to Napster, with the simple query and indexing scheme described in Section 2.1, and where the distributions in the query model are assumed in Section 2.3 to be exponential. In this section, we study the performance of systems that have very different query and system behavior than current music-sharing systems.

Query Model. In Section 2.3, we describe a query model where, given distributions for query frequency and query selectivity, we can calculate the expected number of results for a query, and the expected number of servers needed to satisfy the query. The model itself is completely general in that it makes no assumptions about the type of distributions used for f and g; however, for the purposes of modeling music-sharing systems and validating the model, we made the temporary assumption that f and g are exponential distributions. Implied by this assumption is the additional assumption that f and g are *positively correlated* – that is, the more popular or frequent a query is, the greater the selectivity it has (i.e. the larger the result set).

While we believe that our "music" model can be used in many other domains, one can construct examples where the f and g distributions have different properties. For example, if the system supported complex, expressive relational queries, an obscure query such as `select * from Product where price > 0` would return as many results as a common query such as `select * from Product`. In this case, there is very little, if any, correlation between the popularity of a query and the size of its result set. We say that such systems have *no-correlation* distributions. Another example might be an "archive-driven" system, where users provide old, archived files online but keep their current, relevant information offline. This scenario might occur in the business domain where companies do not want their

	$f(i)$	$g(i)$	Parameters	Correlation	Approx. Pos. Distributions
(a)	$rand(i)/\lambda_f$	$\frac{1}{\lambda_g}e^{-\frac{i}{\lambda_g}}$	$\lambda_f = 100; \lambda_g = 400$	None	$\lambda_f = 100, r = 1$
(b)	$rand(i)/\lambda_f$	$\frac{1}{\lambda_g}e^{-\frac{i}{\lambda_g}}$	$\lambda_f = 1000; \lambda_g = 700$	None	$\lambda_f = 1000, r = 1$
(c)	$\begin{cases} \frac{1}{\lambda_f}e^{-\frac{10000-i}{\lambda_f}} & \text{i} < 10000 \\ \frac{1}{\lambda_f}e^{-\frac{i}{\lambda_f}} & otherwise \end{cases}$	$\frac{1}{\lambda_g}e^{-\frac{i}{\lambda_g}}$	$\lambda_f = 100; \lambda_g = 400$	Negative	$\lambda_f = 100, r = 32$

Table 3.5: Potential Distributions for f and g

current data available to the public or their competitors, but might be willing to provide outdated information for research purposes. Because archives hold historical data, popular queries that refer to recent times return less data, whereas rare queries that refer to the past will return more. In this case, there is a *negative correlation* between query popularity and selectivity.

We now show how distributions that fall into the negative and no-correlation categories can be approximated quite well by distributions in the positive correlation category. Table 3.5 shows three representative f and g distribution pairs from the various categories. The first pair represents the no-correlation category. Distribution g is still exponential, since we know it must be a monotonically decreasing function that sums to 1, and exponential distributions are common in real-life systems. Distribution f is a random function, producing random probabilities falling within the range $[0, \frac{1}{\lambda_f}]$. We choose this range for comparison purposes, since a positively correlated model with the same parameters would have that same range for f. The second pair is similar to the first, except that parameters λ_f and λ_g are different. The third pair is representative of the class of negative correlation distributions. For the top 10000 most popular queries (which accounts for almost 100% of all query occurrences), the more popular the query, the smaller the result set. After the first 10000 queries, we allow the less popular queries to have smaller result sizes in order to account for misspellings, searches for non-existent data, etc. Again, g is exponential, for the same reasons as the first two pairs. Table 3.5 also lists, for each pair, the parameters for a positive correlation distribution that approximates the behavior of the actual negative or no-correlation pair.

Figures 3.9 and 3.10 shows *ExResults* and *ExServ* for each of the representative distributions. *ExResults* is shown in Figure 3.9 as a function of the *total* number of files in the system. In Figure 3.10 *ExServ* is shown as a function of the number of files *per server*, and

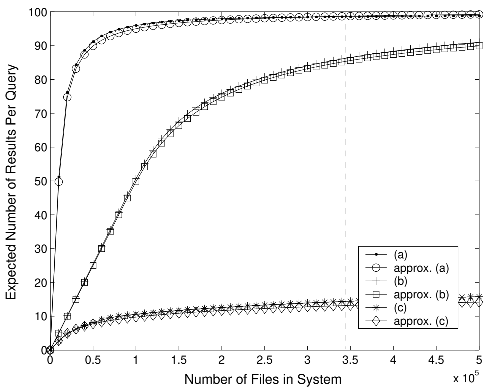

Figure 3.9: *ExTotalResults* for negative and no-correlation parameters

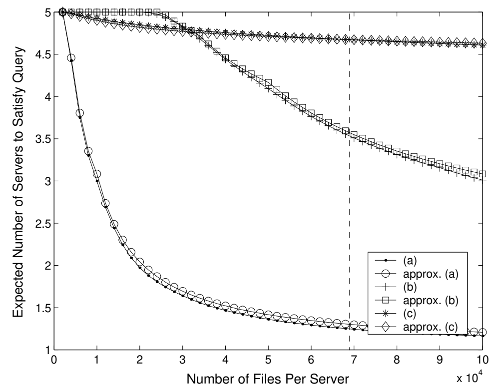

Figure 3.10: *ExServ* for negative and no-correlation parameters

there are assumed to be 5 servers in the system. The dashed lines in both figures show values from the OpenNap system. In addition to showing the curves from the actual negative and no-correlation distributions, Figures 3.9 and 3.10 also show the *approximation curves* derived from the positive correlation approximation distributions listed in Table 3.5. As we can see, the approximations fit the actual curves for *ExResults* and *ExServ* very well. Since the sole purpose of the query model is to calculate values for *ExServ* and *ExResults* given distributions for f and g, we know that overall performance of a system with the negative or no-correlation distributions will be closely matched by a system with the corresponding positive correlation approximation distributions.

Though we can only show a few examples of negative and no-correlation distributions here, additional experimentation has shown that any no-correlation distribution like the ones in Table 3.5, where $\lambda_f = c_f$ and $\lambda_g = c_g$, can be almost exactly approximated by a positive correlation distribution where $\lambda_f = c_f$ and $r = 1$. In addition, though there is no nice formula to describe the relationship, any negative correlation distribution model can be fairly well approximated with a positive correlation distribution where r is very large. This makes sense because positive correlation models with large r have low *ExResults* and therefore high *ExServ*, while negative correlation distributions result in low *ExResults* (and therefore high *ExServ*) as well, since the queries that occur most frequently all return small result sets.

Now that we have shown that distributions with all three types of correlation can be approximated quite closely by positive correlation distributions, by studying the performance of systems with a wide range of positive correlation distributions, we can understand the behavior of systems with negative and no-correlation distributions as well.

Figure 3.11 shows CPU performance of each strategy as r is varied. For comparison, the r values derived for the OpenNap system, and for the approximations of the negative and no-correlation curves, are shown by dotted lines in the figure. First, we notice that increasing r has a much larger effect on the incremental strategies than on batch. For example, at $r = .25$, batch CHN can support 145000 users per server, while incremental CHN can support 110000 users. By the time r reaches 32, however, incremental CHN has seen a 250% improvement in performance, while batch CHN has remained relatively constant. The cause behind this difference is again due to the fact that incremental has poor query performance and good login performance, while batch has poor login performance but good query performance. When r is small, *ExTotalResults* is large, and is limited by *MaxResults*. Queries are thus as expensive as possible, and batch performs better than incremental. In fact, when r increases to 4, the performance of each strategy barely changes because *ExTotalResults* is still equal to *MaxResults*. (However, *ExServ* is increasing. In reality, performance is decreasing very slightly from $r = .25$ to $r = 4$ because of the the extra startup cost at the additional servers, but this decrease is difficult to see in the figure). However, once r increases beyond 4, *ExTotalResults* begins to decrease, first rapidly, then slowly. As a result, the incremental strategies, which up to a certain point were weighed down by slow queries, suddenly improve in performance. The batch strategies, however, which are weighed down by slow logins, barely improve.

In summary, we can make several observations about the performance of strategies in different query models:

- With negative correlation distributions, or positive correlation distributions with high r, incremental strategies are recommended over batch. CPU performance of incremental strategies is usually bound by expensive queries (see Section 3.5.1), but with negative correlation or high r, the expected number of results is very low, thereby making queries inexpensive.

- With no-correlation distributions, or positive correlation with low r, batch strategies outperform incremental (except for FR). This is because when r is low, the expected number of results is very high, thereby making queries expensive and incremental performance poor. Again, batch FR performs so poorly because of expensive logins.

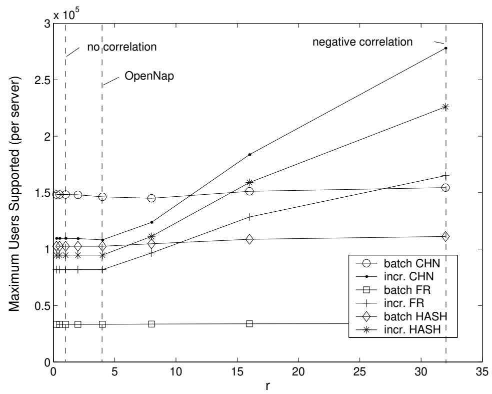

Figure 3.11: CPU Performance as r is varied

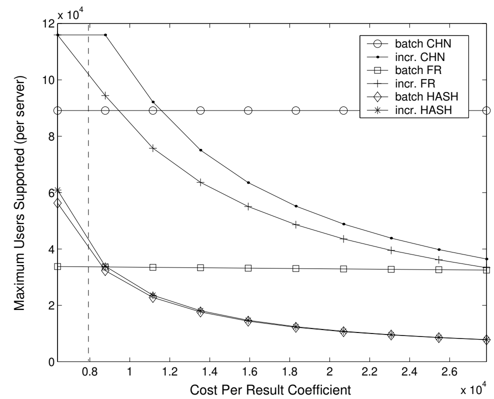

Figure 3.12: Performance as list access coefficient is varied

- For the same reason that incremental shows greater improvement than batch, CHN and HASH show greater improvement than FR as r increases. FR, which has poor login performance but excellent query performance, is least affected by a decrease in *ExResults* caused by an increase in r. Hence, its performance improves the least as r increases.

Performance Model. Unlike the query model described in Section 2.3, the performance model described in Sections 2.4 and 3.4 does make assumptions about the query and indexing scheme used in the system. These assumptions are implied by the *structure* of the cost formulas listed in Table 2.3 (derived in Section 2.4). For example, in Section 2.4 we stated that the CPU cost of a query has the form $c_1 \cdot x + c_2 \cdot y + c_3 \cdot z$, where x is the expected number of results, y is the expected number of remote results, z is the expected number of servers needed to satisfy the query. This formula reflects the assumption that *ExTotalResults*, *ExRemoteResults* and *ExServ* have linear effects on query cost, and that they are the only variables affecting query cost. Clearly, the actual cost of a query in any system would be a much more complex expression involving a number of variables; however, as long as the cost of a query can be approximated well within these assumptions, we can model the system simply by setting coefficients to values appropriate for that system (which we show in the next example). It is only if the cost of processing a query can not be approximated well within these assumptions that the system cannot be analyzed well within our performance model.

For systems that can be described by our performance model, performance depends heavily on the coefficients used in the cost formulas. For example, in Section 2.4 we described how we derived the CPU cost-per-result coefficient for queries in the OpenNap system. However, we can easily imagine how other systems could have very different costs per result. For example, suppose queries consist of a simple key or ID, and the indices are simple hash tables. When a user submits a query, the server simply probes into the hash table with the given ID. In this case, searches would be very inexpensive in terms of CPU cycles – possibly on the order of hundreds of cycles. On the other extreme, suppose the system holds an image database, and queries consist of low resolution images. The system is meant to retrieve all images that are similar to the query image by some comparison technique and criteria. Processing a query in this system would be very expensive because for every result, the server would have to run an expensive image comparison algorithm, and furthermore the cost for all non-matching comparisons would be amortized over the cost for each match. In this case, cost-per-result could be on the order of millions of cycles.

Figure 3.12 shows the effect that varying the cost-per-result coefficient has on overall performance. The coefficient derived in Table 2.3 for the OpenNap/Napster system is 1.1 units (roughly 7800 cycles), marked by a dotted line in the figure. A higher value for the coefficient, for example, 2.2 units (roughly 15556 cycles), means that twice as many cycles are required to both search for and process a single result.

Incremental strategies are most negatively affected by increases in the coefficient, since the negative effect on query performance is magnified by $1/\mathit{ActiveFrac}$ (see Section 3.4). Batch CHN and incremental CHN both start off bound by user-server communication, but as the value for the coefficient increases beyond 9000, incremental CHN becomes CPU-bound, and performance degrades rapidly. Hence for small values of the coefficient, incremental CHN has the best performance, since incremental user-server communication is better than that of batch. However, batch CHN has the best performance when the coefficient is large, because CPU performance is not drastically affected as it is with incremental, and does not become the bottleneck in the range shown. As for the other architectures, we see that once again, FR is CPU-bound and HASH is inter-server communication bound.

In summary, from this experiment we find that:

- For systems with a small cost-per-result coefficient (e.g., OpenNap, hash-lookup systems), incremental strategies are recommended over batch, particularly incremental CHN.

- For systems with large cost-per-result (e.g., image databases), batch CHN is the recommended strategy. Batch performs better when cost-per-result is high because incremental strategies suffer from poor CPU query performance (see Section 3.5.1).

- Batch CHN is also recommended for systems where cost-per-result is unknown or highly variable, since the performance of batch CHN is steady, while even a small increase in cost-per-result causes the performance of incremental CHN to decreases dramatically.

If necessary, all cost coefficients can be varied in the same way in order to view how systems with different query and indexing characteristics perform.

3.6 Conclusion

In this chapter, we studied the behavior and performance of hybrid P2P systems. We developed a probabilistic model to capture the query characteristics of these systems, and an analytical model to evaluate the performance of various architectures and policies. Finally, we evaluated and compared the performance of each strategy. A summary of our findings for each architecture and policy is found below. Unless otherwise noted, performance is measured in terms of the overall efficiency of the system (i.e., the maximum number of users that can be supported).

- In our opinion, the **chained** architecture is the best strategy for today's music-sharing systems, and will continue to be unless network bandwidth increases significantly, or user interests become more diverse. This architecture requires the least amount of memory, and has fast logins that are scalable with respect to the number of servers. However, query performance can be poor if many servers are involved in answering a single query.

- The **full replication** architecture has good potential in the future when network connections will be more stable and memory is cheaper. The architecture performs comparatively well in applications where user interests are diverse, and when result sets are relatively large.

- The **hash** architecture has very high bandwidth requirements but relatively low processing requirements. Hence, it is the best choice only if network bandwidth increases significantly relative to processing power in the future, or in systems where it is not necessary for servers to exchange large amounts of metadata (e.g., systems with search by key or ID, single-word queries).

- The **unchained** architecture is generally not recommended because it returns relatively few results per query and has only slightly better performance than other architectures. It is only appropriate when the number of results returned is not very important compared to the slight performance gain achieved, or when it is the only possible architecture because no inter-server communication is available.
- The **incremental** policy is recommended in systems with negative correlation (e.g., historical or "archive-driven" systems), and has best performance relative to the batch policy when sessions are short and network bandwidth is limited.

Chapter 4

Query Routing in Pure Networks

Pure networks, as described in Chapter 1, refer to P2P networks in which all peers are truly equal in role and functionality. Pure networks have a number of benefits over hybrid ones, including robustness to node failures, self-organization, and even distribution of load. Pure systems are "opportunistic" in that the resource discovery infrastructure is provided by the peers themselves, and no one organization must sustain the cost of maintaining servers. As a result, a large body of research has focused on resource discovery over pure networks.

The best resource discovery techniques over a pure network depend on the needs of the application. For example, for storage or archival systems focusing on availability, *structured* networks (as described in Chapter 1) such as Chord [85], Pastry [78], Tapestry [99] and CAN [74] are well-suited, because they guarantee location of content if it exists, within a bounded number of hops. To achieve these properties, these techniques tightly control the data placement and topology within the network, and currently only support search by object identifier.

In systems where persistence and availability are not guaranteed or necessary, such as Gnutella [35], Freenet [30], Napster [62] and Morpheus [61], search techniques can afford to have looser guarantees. However, because these systems are meant for a wide range of users that come from non-cooperating organizations, the techniques cannot afford to strictly control the data placement and topology of the network. Friends may want to connect to other friends, while strangers may not want to store data (potentially large amounts) on behalf of each other. Also, these systems traditionally offer support for richer queries than just search by identifier, such as keyword search with regular expressions. Search techniques for these "loose," unstructured systems must therefore operate under a different set of constraints than techniques developed for highly structured applications.

Current search techniques in unstructured pure P2P systems tend to be very inefficient, either generating too much load on the system, or providing for a very bad user experience (see Section 4.2 for a discussion). In this chapter, we present the design and evaluation of new search techniques for loosely controlled, loose guarantee systems such as Gnutella and Morpheus. In particular, our main contributions are:

- We present several search techniques that achieve huge performance gains over current techniques, but are simple and practical enough to be easily incorporated into existing systems (Section 4.3).

- We evaluate our techniques using large amounts of data gathered from Gnutella, the largest open P2P network in operation (Section 4.4).

- We highlight the strengths of each technique as well as the weaknesses, and translate these tradeoffs into practical recommendations for today's systems (Section 4.5).

As we will see, the techniques presented in this chapter adhere to the design goals laid out in Chapter 1 – they are simple and practical enough to be easily incorporated into existing systems, and they are adaptive on a user and query level.

The basic idea behind our techniques is to reduce the number of nodes that receive and process each query. Doing so reduces the aggregate load generated by each query across the network. If we assume that each node can only answer queries about their own content, then naturally, the fewer nodes that process the query, the fewer results that will be returned. Our techniques will only be effective if most queries can be answered by querying fewer nodes than simple flooding techniques. Indeed, past work in [94], and our own experiments in Section 4.5, show that most queries can be answered by querying fewer nodes than the current techniques. Hence, our first technique, *Iterative Deepening* (Section 4.3.1), reduces the number of nodes that are queried by iteratively sending the query to more nodes until the query is answered. The *Directed BFS* technique (Section 4.3.2), queries a restricted set of nodes that are intelligently selected to maximize the chance that the query will be answered.

If instead we allow nodes to answer queries on behalf of other nodes, then we can still reduce the number of nodes that process a query without relying on the ability of a few nodes to answer most queries. In the *Local Indices* technique (Section 4.3.3), nodes maintain very simple and small indices over other nodes' data. Queries are then processed by a smaller set of nodes than current techniques, while yielding as many results.

The rest of this chapter is organized as follows. In Section 4.1 we give an overview of related work in the area. Section 4.2 gives an overview of the search problem, presents metrics with which to evaluate techniques, and discusses the shortcomings of techniques used in today's systems. Then in Section 4.3, we present our three search techniques, describing in detail the different variations and policies within each technique. These techniques and variations are then evaluated in Section 4.5, using the experiments and analysis described in Section 4.4.

Many contributions in this chapter originally appeared in [95].

4.1 Related Work

In addition to the work presented here, other search techniques for unstructured systems include [1, 17, 64]. The search technique proposed in [1] is similar to our Local Indices technique (Section 4.3.3), where nodes index the content of other nodes in the system, thereby allowing queries to be processed over a large fraction of all content while actually visiting just a small number of nodes. However, the routing policy for Query messages differ between our work and theirs, resulting in different performance and cost tradeoffs.

References [17] and [64] propose that each node maintain metadata that can provide "hints" as to which nodes contain data that can answer the current query. Query messages are routed by nodes making local decisions based on these hints. Hints in [17] are formed by building *summaries* of the content that is reachable via each connection a node has. Hints in [64] are formed by learning user preferences. By observing the behavior of the user – for example, noting which nodes the user chose to download from – a user's client can learn which nodes the user believes to be an *authority* on certain topics. Future queries on those topics can subsequently be routed to these authority nodes.

Search techniques for structured networks with strong guarantees on availability include Chord [85], CAN [74], Pastry [78] and Tapestry [99], as described in Chapter 1. These four techniques are quite similar in concept, but differ slightly in algorithmic and implementation details. In Chord, Pastry and Tapestry, nodes are assigned a numerical identifier, while in CAN, nodes are assigned regions in a d-dimensional identifier space. A node is then responsible for owning objects, or pointers to objects, whose identifiers map to the node's identifier or region. Nodes also form connections based on the properties of their identifiers. With these deliberately formed connections and intelligent routing, a system can locate an object by its identifier within a bounded number of hops. These techniques perform well for the systems they were intended for [19, 53, 79], but we reiterate that they may not be appropriate for the type of system we are interested in studying, and are not as widely used.

4.2 Problem Overview

Recall that in our file-sharing model (presented in Section 2.1), each node owns a collection of files to be shared with other nodes, and queries are a list of desired words to appear in a file's metadata. Note, however, that the techniques presented in this chapter do not require that the shared data consist of files. For example, the shared data could consist of records stored in a relational database. Queries may take any form that is appropriate given the type of data shared, such as SQL queries over database records.

In a pure P2P system, peers form connections with other peers to create an overlay network, as described in Chapter 1. We can view an overlay network in this context as an undirected graph, where the vertices correspond to nodes in the network, and the edges correspond to open connections maintained between the nodes. Two nodes maintaining an open connection between themselves are known as *neighbors*. The number of neighbors a node has is called its *outdegree*. Messages may be transferred in either direction along the edges. For a message to travel from node A to node B, it must travel along a path in the graph. The number of edges along this traveled path is known as the message *hop count*, or alternatively, its *path length*. Similarly, two nodes are said to be "n hops apart" if the shortest path between them has length n. Often messages are given a maximum *time to live*, or TTL, which specifies the maximum number of hops the message may take before it must be dropped.

Joins, leaves and updates are simple in a basic pure system (variants on the pure architecture, such as those described in Section 4.3.3 and Chapter 5, may render these actions more complex). When a user joins the system, her node simply forms connections with a handful of other nodes. When the user leaves, her node simply drops these connections. When an update is made to a file or its metadata at a particular peer, no action is required at any other peer.

Queries are the most complex and expensive actions in a pure system. When a user submits a query, her node becomes the *source* of the query. A source node S may send the query message to any number of its neighbors. The routing policy in use determines to how many neighbors, and to which neighbors, the query is sent. When a node receives a Query message, it will process the query over its local collection. If any results are found at that node, the node will send a single Response message back to the query source. In some systems, such as Gnutella, the address of the query source is unknown to the replying node. In this case, the replying node sends the Response message along the reverse path traveled by the query message. In other systems, the replying node may know the address of the query source, and will open a temporary connection with the source to transfer the Response message. Although the first scheme uses more aggregate bandwidth than the second, it provides anonymity for the query source, and it prevents the query source from being bombarded by connection requests. In this chapter, we assume the first scheme is being used.

When a node receives a Query message, it must also decide whether to forward the message to other neighbors, or to drop it. Again, the routing policy determines whether to forward the query, and to whom the query is forwarded.

Metrics. To evaluate the performance of our techniques, we measure average aggregate bandwidth and processing power cost incurred by queries, joins/leaves and updates. We also measure quality of results using number of results, satisfaction rate, and response time. More specifically, response time is defined as the time to satisfaction. Given that Z results are required to satisfy a query, time to satisfaction is defined as the time that has elapsed from when the query is first submitted by the user, to when the user's peer receives the necessary number of results for satisfaction. For further details on metrics, please refer to Section 2.5.

Existing Techniques. Consider the search techniques used by the existing Gnutella and Freenet systems:

- **Gnutella**: Uses a breadth-first traversal (BFS) over the network with depth limit D, where D is the system-wide maximum time-to-live of a message, measured in hops. A source node S sends the Query message with a TTL of D to a depth of 1, to all its neighbors. Each node at depth 1 processes the query, sends any results to S, and then forwards the query to all of their neighbors, to a depth of 2. This process continues until depth D is reached (i.e., when the message reaches its maximum TTL), at which point the message is dropped.

- **Freenet**: Uses a depth-first traversal (DFS) with depth limit D. Each node forwards the query to a single neighbor, and waits for a definite response from the neighbor before forwarding the query to another neighbor (if the query was not satisfied), or forwarding results back to the query source (if the query was satisfied).

If the quality of results in a system were measured solely by the number of results, and not by satisfaction, then the BFS technique is ideal because it sends the query to every possible node (i.e., all nodes within D hops), as quickly as possible. However, if satisfaction were the metric of choice, BFS wastes much bandwidth and processing power because, as we have argued earlier, most queries can be satisfied from the responses of relatively few nodes. With DFS, because each node processes the query sequentially, searches can be terminated (i.e., the query message dropped) as soon as the query is satisfied, thereby minimizing cost. However, sequential execution also translates to poor response time, with the worst case being exponential in D. Actual response time in Freenet is moderate, because only a single result is returned per query (as opposed to many results in Gnutella), and simple indexes are used to enable more intelligent routing.

4.3 Broadcast Policies

As we can see, existing techniques fall on opposite extremes of bandwidth/processing cost and response time. Our goal is to find some middle ground between the two extremes, while maintaining quality of results.

4.3.1 Iterative Deepening

In systems where satisfaction is the metric of choice, a good technique is *Iterative Deepening*. Iterative Deepening is a well-known search technique used in other contexts, such as search over state space in artificial intelligence [80].

Over the iterations of the Iterative Deepening technique, multiple breadth-first searches are initiated with successively larger depth limits, until either the query is satisfied, or the maximum depth limit D has been reached. Because the number of nodes at each depth grows exponentially (Section 4.4.1), the cost of processing the query multiple times at small depths is small, compared to processing a query once at a large depth. In addition, if we can satisfy the query at a depth less than D, then we can use much fewer resources than a single BFS of depth D. Indeed, our experiments show (Section 4.4.1) that for a client with 8 neighbors in the Gnutella network where $D = 7$, 70% of all queries can be satisfied at a depth less than D, for $Z = 20$.

The Iterative Deepening technique is implemented as follows: first, a system-wide *policy* is needed, that specifies at which depths the iterations are to occur. For example, say we want to have three iterations: the first iteration searches to a depth a, the second to depth b, and the third at depth c. Our policy is therefore $P = \{a, b, c\}$. Note that in order for Iterative Deepening to have the same performance as a BFS of depth D, in terms of satisfaction, the last depth in the policy must be set to D. In addition to a policy, a waiting period W must also be specified. W is the time between successive iterations in the policy, explained in further detail below.

Under the policy $P = \{a, b, c\}$, a source node S first initiates a BFS of depth a by sending out a Query message with TTL$= a$ to all its neighbors. Once a node at depth a receives and processes the message, instead of dropping it, the node will store the message temporarily. The query therefore becomes "frozen" at all nodes a hops from the source S. Nodes at depth a are known as the *frontier* of the search. Meanwhile, S receives Response messages from nodes that have processed the query so far. After waiting for a time period W, if S finds that the query has already been satisfied, then it does nothing. Otherwise, if the query is not yet satisfied, S will start the next iteration, initiating a BFS of depth b.

To initiate the next BFS, S could send out another Query message with TTL$= b$, meaning every node within a hops will process the query a second time. To prevent this wasted effort, however, S will instead send out a *Resend* message with a TTL of a. Instead of reprocessing the query, a node that receives a Resend message will simply forward the message, or if the node is at the frontier of the search, it will drop the Resend message and "unfreeze" the corresponding query by forwarding the Query message (with a TTL of $b - a$) to its neighbors. To identify queries with Resend messages, every query is assigned a system-wide "almost unique" identifier. Gnutella already assigns such an identifier to all queries. The Resend message will contain the identifier of the query it is representing, and nodes at the frontier of a search will know which query to unfreeze by inspecting this identifier. Note that a node need only freeze a query for slightly more than W time units before deleting it.

After the search to depth b, the process continues in a similar fashion to the other levels in the policy. Since c is the depth of the last iteration in the policy, queries will not be frozen at depth c, and S will not initiate another iteration, even if the query is still not satisfied.

4.3.2 Directed BFS

If minimizing response time is important to a particular application, then the Iterative Deepening technique may not be applicable because of the time taken by multiple iterations. A better strategy that still reduces cost would be to send queries immediately to a subset of nodes that will return many results, and will do so quickly.

The Directed BFS (DBFS) technique implements this strategy by having a query source send Query messages to just a subset of its neighbors, but selecting neighbors through which nodes with many quality results may be reached. For example, one may select a neighbor that has produced or forwarded many quality results in the past, on the premise that past performance is a good indication of future performance. The neighbors that receive the query then continue forwarding the message to all neighbors as with BFS.

In order to intelligently select neighbors, a node will maintain statistics on its neighbors. These statistics can be very simple, such as the number of results that were received through the neighbor for past queries, or the latency of the connection with that neighbor. From these statistics, we can develop a number of heuristics to help us select the best neighbor to send the query. Sample heuristics include:

- Select the neighbor that has returned the highest number of results for previous queries.

- Select neighbor that returns response messages that have taken the lowest average number of hops. A low hop-count may suggest that this neighbor is close to nodes containing useful data.

- Select the neighbor that has forwarded the largest number of messages (all types) since our client first connected to the neighbor. A high message count implies that this neighbor is stable, since we have been connected to the neighbor for a long time, and it can handle a large flow of messages.

- Select the neighbor with the shortest message queue. A long message queue implies that the neighbor's pipe is saturated, or that the neighbor has died.

By sending the query to a small subset of neighbors, we will likely reduce the number of nodes that receive the query by a significant amount, thereby reducing costs incurred by the query. On the other hand, by selecting neighbors that we believe will produce many results, we can maintain quality of results to a large degree, even though fewer nodes are visited. In our experiments, a query source sends the query to a single neighbor only. Surprisingly, we will see that the quality of results does not decrease significantly, provided that we make intelligent neighbor selections.

Variations. Several variations of Directed BFS are possible. We discuss some options here, but do not evaluate them experimentally.

First, rather than send the query to a single neighbor, we may send the query to a subset of neighbors. Increasing the size of the subset may increase the quality of results, but will also increase the cost. The degenerate case where the query is sent to all neighbors is simply BFS.

Second, we observe that Directed BFS is really a hybrid between DFS and BFS. The first hop is depth-first, since we forward the query to a single neighbor. The remaining hops are breadth-first. As another variation on the technique, we can explore the possibility of making the first n hops depth-first. For example, if $n = 2$, then a client will forward a query to a single neighbor, who will then select one of its neighbors to forward the query to. That neighbor will then perform BFS at a depth of $D - 2$. In this way, we have finer control over the size of the population we perform costly BFS over, since the larger n is, the smaller the depth at which BFS is performed. One drawback of this variation is increased response time. As we increase n, we get closer to true DFS, which as we mentioned, has bad response time. Hence, we again encounter the tradeoff between response time and bandwidth. This time, however, by varying n we can explore the continuum between the two extremes: BFS ($n = 0$) and DFS ($n = 7$).

4.3.3 Local Indices

In the Local Indices technique, a node n maintains an index over the data of each node within r hops of itself, where r is a **system-wide** variable known as the *radius* of the index ($r = 0$ is the degenerate case, where a node only indexes metadata over its own collection).

When a node receives a Query message, it can then process the query on behalf of every node within r hops of itself. In this way, the collections of many nodes can be searched by processing the query at few nodes, thereby maintaining a high satisfaction rate and number of results while keeping costs low.

Previously, we mentioned that data placement strategies may face a barrier to implementation in loosely controlled systems such as Gnutella, because users will not want to devote large amounts of storage to store other users' data or metadata. However, when r is small, the amount of metadata a node must index is also quite small – on the order of 50 KB – independent of the total size of the network. As a result, Local Indices with small r should be easily accepted by a loosely controlled system such as Gnutella. See Section 4.5.3 for more details on space requirements.

Before we describe details of this technique, let us first point out that the concept of indexing other nodes' data has been used in several systems to date. We mentioned earlier that the Morpheus P2P system uses "super-peers" – nodes that index the collections of their clients and answer queries on their behalf, while the clients never answer any queries. Napster can be seen as using a variant of the super-peer technique, where a server containing a centralized index over every node's data is the single super-peer, and all other nodes are clients. Our Local Indices technique differs from these other techniques because all nodes under this technique still have equal roles and capabilities, i.e., it is still pure P2P. Furthermore, our technique is malleable in that we can adjust r to study both systems with small indices (small r) such as Morpheus, systems with large indices (large r) such as Napster, and many possibilities in between. As a result, it is well-suited to understanding the benefits and tradeoffs of indexing.

The Local Indices technique works as follows: a policy specifies the depths at which the query should be processed. All nodes at depths not listed in the policy simply forward the query to the next depth. For example, say the policy is $P = \{1, 5\}$. Query source S will send the Query message out to its neighbors at depth 1. All these nodes will process the query, and forward the Query message to all their neighbors at depth 2. Nodes at depth 2 will not process the query, since 2 is not in the policy P, but will forward the Query message to depth 3. Eventually, nodes at depth 5 will process the query, since depth 5 is in the policy. Also, because depth 5 is the last depth in P, these nodes will then drop the Query message. We assume that in a given system, all queries use the same policy.

Note the difference between a Local Indices policy and an Iterative Deepening policy, where depths in the policy represent the depths at which iterations should end, and nodes at *all* depths process the query. Note also that in order for the Local Indices technique to have equal performance as a BFS of depth D, in terms of number of results and satisfaction, the last depth in the Local Indices policy must be set to $D - r$.

To create and maintain the indices at each node, extra steps must be taken whenever a node joins or leaves the network, and whenever a user updates his local collection (e.g., inserts a file). As described in Section 2.1, when a node X joins the network, it sends a Join message with a TTL of r, which will be received by all nodes within r hops. The Join message contains sufficient metadata over X's collection for another node to answer queries on node X's behalf. When a node receives the Join message from X, it will in return send a Join message containing metadata over its collection directly to X (i.e., over a temporary connection). Both nodes then add each other's metadata to their own index.

When a node joins the network, it may introduce a path of length r or less between two other nodes, where no such path previously existed. In this case, the two nodes can be made aware of their new connection in a number of ways without introducing additional messages. For example, in the Gnutella network, nodes periodically send out Ping messages to all nodes within a depth D, and every node receiving the Ping should respond with a Pong message. This mechanism is used for node discovery. If a node A receives a Pong message from another node B that is within r hops, but whose metadata is not currently indexed at A, node A may send a Join message directly to B, and in return, B may send a Join message directly to A.

When a node leaves the network or dies, other nodes that index the leaving node's collection will remove the appropriate metadata after a timeout. Timeouts can also be implemented without introducing new messages. For example, if a node does not respond to Ping messages several times, it may be assumed that this node has died or left the network.

When a user updates his collection, his node will send out a small Update message with a TTL of r, containing the metadata of the affected data element (e.g., file or record), and indicates whether the element was inserted, deleted, or updated. All nodes receiving this message subsequently update their index.

To translate the cost of joins, leaves and updates to query performance, we amortize these costs over the cost of queries. The parameter `QueryJoinRatio` gives us the average ratio of queries to joins in the entire P2P network, while `QueryUpdateRatio` gives us the average ratio of queries to updates (see Table 2.1).

Variations. In the future, we can incorporate techniques from Iterative Deepening with Local Indices. For example, if $r = 1$, and most queries can be satisfied by results coming from nodes within a radius of 6, it might be worthwhile to implement a policy such as $p = \{2, 5, 6\}$. The query will be executed at levels 2 and 5, which will cover the content of all nodes within a radius of 6. If the query is satisfied, which will usually be the case, then the query is done. Otherwise, Resend packets can be sent out to level 5, and the query processed at level 6, in order to cover the nodes at level 7.

In our experiments, we do not evaluate policies where Iterative Deepening and Local Indices are combined.

4.4 Experimental Setup

We chose to base our evaluations on data gathered from the Gnutella network because Gnutella is the largest open P2P system in operation, with about 50000 users as of May, 2001. To evaluate our techniques, we need to answer questions such as:

- When our Gnutella client submits a query, how many results are received?
- When our client submits a query, what is the time to satisfaction under the Iterative Deepening technique?

Some of these questions can be answered through direct experiments; for example, we can answer the first question simply by having our client parse and log each Response message it gets in response to its query. Other questions need to be answered indirectly; for example, we could not force all Gnutella nodes to use the Iterative Deepening technique, in order to answer our second question. Instead, we collect performance data for queries under the BFS technique, and combine this data using analysis to estimate what the time to satisfaction would have been under Iterative Deepening. We therefore evaluate our techniques using a combination of experiments to gather raw data from the Gnutella network, and analysis to estimate the cost and performance of queries under our techniques. Experiments and analysis are described in Sections 4.4.1 and 4.4.2, respectively.

Note that the evaluations performed in this chapter do not follow the model presented in Chapter 2, due to the availability of trace data that allows us to evaluate our techniques in a more realistic manner.

4.4.1 Data Collection

First, we needed to gather some general information on the Gnutella network and its users. For example, how many files do users share? What is the typical size of metadata for a file? The experimental process to gather these general statistics is described in Section 2.2. During this process, our client also passively observed the query strings of query messages that passed through the network. To get a representative set of queries for Gnutella, Q_{rep}, we randomly selected 500 queries from the 500,000 observed queries.

Iterative Deepening

From the representative set Q_{rep}, our client submitted each query Q to the Gnutella network D times (spread over time), where $D = 7$ is the maximum TTL allowed in Gnutella. Each time the query was submitted we increment its TTL by 1, so that we submit the query once for each TTL between 1 and D. For each Query message submitted, we logged every Response message that arrived within 2 minutes of submission of the query. For each Response, we log:

- The number of hops that the Response message took.
- The response time (i.e., the time elapsed from when the Query message was first submitted, to when the Response message was received).
- The IP address from which the Response message came.
- The individual results contained in the Response message.

As queries are submitted, our client sent out Ping messages to all its neighbors. Ping messages are propagated through the network in a breadth-first traversal, as Query messages are. When a node receives a Ping message, it replies with a Pong message containing its IP (and other information). We sent a Ping message immediately before every second query. After a Ping message was sent, for the duration of the next two queries (i.e., 4 minutes), we logged the following information for all Pong messages received:

- The number of hops that the Pong message took.
- The IP address from which the Pong came.

Symbol	Description
$L(Q)$	Length of query string for query Q
$M(Q, n)$	Number of response messages received for query Q, from n hops away
$R(Q, n)$	Number of results received for query Q, from n hops away
$S(Q, n, Z)$	Returns true if query Q received Z or more results from n hops away

Table 4.1: Symbols and function names of data extracted from logs for Iterative Deepening

Symbol	Description
$T(Q, Z, W, P)$	Time to satisfaction of query Q, under Iterative Deepening policy P and waiting time W
$N(Q, n)$	Number of nodes n hops away that process Q
$C(Q, n)$	Number of redundant edges n hops away

Table 4.2: Symbols and function names of data estimated from logs for Iterative Deepening

From these Response and Pong logs, we can directly extract information necessary to estimate the cost and quality of results received for each query, summarized in Table 4.1. Each of these data elements were extracted for every query $Q \in Q_{rep}$, and for every possible hop value n ($n = 1$ to D). In the definitions of $R(Q, n)$ and $M(Q, n)$, note that a single Response message can hold more than one result record. Binary function $S(Q, n, Z)$ returns true if query Q was satisfied by the results received from nodes within n hops.

In addition to the values we can extract directly from our logs, we also estimate several values that could not be directly observed by our client, listed in Table 4.2. Though these values could not be directly observed, they are nevertheless carefully calculated from observed quantities, as follows.

Estimating $N(Q, n)$. Under BFS, if a Ping and Query message are sent out at the same time from the same node with the same TTL, in theory, every node that processes the Query message should also process the Ping. Hence, the number of nodes that process the Query should be equal to the number of nodes that return a Pong message. In practice, however, not all nodes that process the query will also return a Pong message. For example, some nodes do not respond to Ping messages, to save bandwidth. To estimate $N(Q, n)$, therefore, we look at the following sets of IP addresses: (1) Set A: The IP addresses of nodes n hops away that responded to the latest Ping sent out before query Q was submitted, (2) Set B: The IP addresses of nodes n hops away that responded to the earliest Ping sent out after query Q was submitted, and (3) Set C: The IP addresses of nodes n hops away that responded to query Q. All three sets of nodes generally overlap, and we know that the "correct" set of nodes contains some subset of A and B, and completely contains C. Since nodes join and leave the network frequently, using the approximation $N(Q,n) = |A \cup B \cup C|$ will almost certainly be an over-estimation. However, using $|A \cup C|$, which is theoretically the correct choice, will almost certainly be an under-estimation since many nodes do not respond to Pings. Hence, we estimate $N(Q,n)$ to be $N(Q,n) = MAX(|A \cup C|, |B \cup C|)$.

Estimating $C(Q,n)$. An edge is said to be *redundant* at depth n if a message traveling through this edge on its nth hop arrives at a node that has already seen the message – that is, if it closes a cycle on the nth hop. To estimate $C(Q,n)$, the number of redundant edges at depth n, we observed in an earlier experiment that our client dropped roughly 30% of the Query messages it received. A Query message is dropped if it has been seen by the client before – that is, if it has just traveled through a redundant edge. Given that our client dropped 30% of all Query messages it received, and assuming our client to be a model for typical behavior, we estimate 1 out of 3 of all messages in a breadth-first traversal are dropped due to redundant edges. In other words, if there are 60 nodes at depth 3, then 90 messages are sent from nodes at depth 2 to nodes at depth 3, and 30 of these messages are dropped. Since $N(Q,n)$ represents the number of messages that are not dropped, $C(Q,n) = N(Q,n)/2$.

Estimating $T(Q,Z,W,P)$. Let $L_{(Q,n)}$ denote the ordered list of times at which result records are received for query Q from nodes n hops away. For example, $L_{(Q,n)}[i]$ is the time the ith result record was received from a node n hops away. The time at which a result record is received is simply the time elapsed from when the query is first submitted, to when the response message containing that record was received. From our logs, we can directly extract $L_{(Q,n)}$ for each query Q, for each value of n from 1 to D.

Now let us define $L^t_{(Q,n)}$ to be a list of the same length as $L_{(Q,n)}$, and where $L^t_{(Q,n)}[i] = L_{(Q,n)}[i] + t$. That is, we increment every element in $L_{(Q,n)}$ by t. Let us define function `merge(L,K)` over two ordered lists L and K to return a single ordered list formed by merging the elements of list L and K. Finally, let us define function $I(n,P)$ to return the iteration on which depth n will be reached, under policy P. For example, if $P = \{2,5,7\}$, then $I(1,P) = 1$, $I(2,P) = 1$, $I(3,P) = 2$, etc.

We can now calculate the time to satisfaction for a query Q by the following procedure:

$TTS(Q, P)$

(1) $T = L_{(Q,1)}$

(2) for $n = 2$ to D

(3) $T = \text{merge}(T, L_{(Q,n)}^{(I(n,P)-1)\cdot W})$

(4) return $T[Z]$

Let us assume query Q does not end before reaching depth n under policy P. In this case, then nodes at depth n will receive the query on the $I(n, P)$th iteration, meaning the nodes would not receive the query until $(I(n, P) - 1) \cdot W$ time units after the query was first submitted. We therefore estimate the times at which these results arrive at the source node to be $L_{(Q,n)}^{(I(n,P)-1)\cdot W}$.

Let us assume that query Q is not satisfied until the last iteration. Then, the response times at which each result, for all depths, are received is given by T, calculated in lines $1-3$ in the above procedure. The times at which the query is satisfied is the time at which the Zth result is received, which is $T[Z]$.

Now, assume query Q is satisfied at some iteration i that is not the last iteration. Let x be the depth of the iteration $i+1$. All depths d such that $d \geq x$ will not be reached by query Q. Will merging $L_{(Q,d)}^{(I(d,P)-1)\cdot W}$ into T for $d \geq x$ therefore give us an incorrect result? Since Q is satisfied during iteration i, we know that it satisfied at some time prior to $(I(n, P) - 1) \cdot W$, by definition (otherwise, iteration $i + 1$ would have been initiated). Hence, there are at least Z elements in T which are less than $(I(n, P) - 1) \cdot W$. Since merging $L_{(Q,d)}^{(I(d,P)-1)\cdot W}$ into T for $d \geq x$ will only add elements into T that are greater or equal to $(I(n, P) - 1) \cdot W$, the value of $T[Z]$ is unaffected by the merging of these lists. Thus, the above procedure returns the correct time to satisfaction for query Q.

Directed BFS

The experiments for DBFS are similar to the experiments for Iterative Deepening, except each query in Q_{rep} is now sent to a single neighbor at a time. That is, rather than sending the same Query message, with the same message ID, to all neighbors, our node sends a query message with a different ID (but same query string) to each neighbor. Similarly, Ping messages with distinct IDs are also sent to a single neighbor at a time, before every other query.

For each Response and Ping received, our client logs the same information logged for Iterative Deepening, in addition to the neighbor from which the message is received. From

Symbol	Description
$L(Q)$	Length of query string for query Q
$M(Q, n, y)$	Number of response messages received for query Q, from nodes n hops away, when the Query message was sent to neighbor y
$R(Q, n, y)$	Number of results received for query Q, from nodes n hops away, when the Query message was sent to neighbor y
$S(Q, n, Z, y)$	Returns true if query Q receives Z or more results from n hops away, when the Query message was sent to neighbor y
$T(Q, Z, y)$	The time at which query Q is satisfied, when the query was sent to neighbor y

Table 4.3: Symbols and function names of data extracted from logs for Directed BFS

our logs, we then extract the same kind of information as with Iterative Deepening. However, now we differentiate the performance of the query when sent to different neighbors. The data we extract from the logs is listed in Table 4.3. Note that unlike with Iterative Deepening, the time to satisfaction in Directed BFS – $T(Q, Z, y)$ – is directly observable by our client. We also estimate several values that could not be directly extracted, listed in Table 4.4. These values are estimated in the same manner as described in the previous section on Iterative Deepening.

In addition to gathering Response and Pong information, we also recorded the following statistics for each neighbor right before each query was sent out:

- Number of results for previous queries from this neighbor, with a maximum query history of 10
- Average TTL of results for previous queries from this neighbor, with a maximum query history of 10
- Average response time of results for previous queries from this neighbor, with a maximum query history of 10
- Number of messages (all types) received from this neighbor, since our client first connected to the neighbor
- Number of messages (all types) sent to this neighbor, since our client first connected to the neighbor
- Number of messages (all types) dropped from this neighbor, since our client first connected to the neighbor
- Size of the message queue to this neighbor, at the time the query is submitted

Symbol	Description
$N(Q,n,y)$	Number of nodes n hops away that process Q when the Query message was sent to neighbor y
$C(Q,n,y)$	Number of redundant edges n hops away when the Query message was sent to neighbor y

Table 4.4: Symbols and function names of data estimated from logs for Directed BFS

Symbol	Value (Bytes)	Description
$a(Q)$	$82 + L(Q)$	Size of a Query message
b	80	Size of a Resend message
c	76	Size of a single result record
d	80 + 28	Size of a Response message header
$e(Q,r)$	$80 + 28\cdot$ # addr $+76\cdot$ # results	Size of a full Response message under Local Indices
f	$80 + 72\cdot$ # files	Size of a Join message
g	152	Size of an Update message

Table 4.5: Sizes of messages

- Latency of pong responses from this neighbor, at the time the query is submitted
- $N(Q,1,y)$: an estimate of the degree of the neighbor immediately before query Q is submitted

These statistics are then used in heuristically selecting to which neighbor to forward the query.

Local Indices

No separate experiment was run to gather data for this technique. Instead, we use the data from the Iterative Deepening and Directed BFS experiments.

4.4.2 Calculating Costs

Given the data we collect from the Gnutella network, we can now estimate the cost and performance of each technique through analysis.

Bandwidth Cost

Message sizes for the Gnutella protocol are discussed in Section 2.4, and repeated in Table 4.5 for reader convenience. Note that for f, the size of a Join message, we use the average number of files observed in the Gnutella network for the "# files" parameter. As shown in Table 2.1, this value was observed to be roughly 340 files. Likewise, under Local Indices, a peer will return responses that belong to all nodes within r hops of itself – those nodes covered by the local index at the peer. To estimate the number of responses (# files), we use the number of results our own node received from nodes within r hops, and scaled this value appropriately to reflect the fact that our node had x neighbors at the time Q was submitted, but the average degree of a node is roughly 3. Likewise, we estimate the number of nodes to which these responses belong (# addr) by using the number of nodes within r hops of our node that replied to query Q, scaled appropriately.

Given message sizes and logged information from our trace experiments, we can estimate aggregate bandwidth consumed by a single query under the various techniques. For example, aggregate bandwidth for a query under BFS is:

$$BW_{bfs}(Q) = \sum_{n=1}^{D} \left(a(Q) \cdot \Big(N(Q,n) + C(Q,n) \Big) + n \cdot \Big(c \cdot R(Q,n) + d \cdot M(Q,n) \Big) \right) \tag{4.1}$$

We can understand the above formula by thinking of the query as it travels from one depth to the next – that is, when it travels from n hops away to $n+1$ hops away. First, we know that the query will reach every level from 1 to D, which is why we have the summation from $n = 1$ to D. Here, n represents the depth, or number of hops the query has taken. Recall that D represents the maximum TTL of a message.

The first term inside the summation gives us the bandwidth consumed by sending the query message from level $n-1$ to level n. There are $N(Q,n)$ nodes at depth n, and there are $C(Q,n)$ redundant edges between depths $n-1$ and n. Hence, the total number of Query messages sent on the nth hop is equal to $N(Q,n) + C(Q,n)$. If we multiply this sum by $a(Q)$, the size of the Query message, we get the bandwidth consumed by forwarding the query on the nth hop.

The second term gives us the bandwidth consumed by transferring Response messages from n hops away back to the query source. There are $M(Q,n)$ Response messages returned from nodes n, and these Response messages contain a total of $R(Q,n)$ result records. The size of a response message header is d, and the size of a result record is c (on average), hence the total size of all Response messages returned from n hops away is $c \cdot R(Q,n) + d \cdot M(Q,n)$. These messages must take n hops to get back to the source; hence, bandwidth consumed is n times the size of all responses.

Formulae for calculating aggregate bandwidth consumption for the remaining policies –

Symbol	Definition
D	Maximum time-to-live of a message, in terms of hops
Z	Number of results needed to *satisfy* a query
Q_{rep}	Representative set of queries for the Gnutella network

Table 4.6: Definition of Symbols

Iterative Deepening, Directed BFS and Local Indices – follow the same pattern, and include the same level of detail, as Equation 4.1. For reader clarity, the formulae and derivations are not included here, but can be found in Appendix C.1.

Processing Cost

Calculating average aggregate processing power consumed under a particular technique is done much in the same way as calculating average aggregate bandwidth. First, we estimate how much processing power each type of action requires, described in Section 2.4. We then go through the data in the query logs and use analysis to estimate how much aggregate processing power each query consumed. Again, for reader clarity, formulae for calculating aggregate processing cost is given in Appendix C.2.

4.4.3 Averaging Results

Now that we can calculate or directly observe the performance of a single query Q in terms of each metric, the performance of a technique is measured by averaging the performance of every query $Q \in Q_{rep}$. For example, average aggregate bandwidth for a breadth-first search is:

$$AvgAggBW_{bfs} = \frac{1}{|Q_{rep}|} \cdot \sum_{Q \in Q_{rep}} BW_{bfs} \tag{4.2}$$

4.5 Experiments

In this section, we present the results of our experiments and analysis. We discuss the performance tradeoffs between different policies, and form recommendations for existing systems such as Gnutella. As a convenience to the reader, some symbols defined in previous sections are re-defined in Table 4.6.

Note that our evaluations are performed over a single "real" system, and that results may vary for other topologies. Nevertheless, since Gnutella does not control topology or data placement, we believe its characteristics are representative of the type of system we want to study.

4.5.1 Iterative Deepening

For the sake of comparison, we evaluate only those Iterative Deepening policies where the last depth is set to D (see Section 4.3.1 for an explanation). To understand the tradeoffs between policies of different lengths, and policies that begin at different depths, we choose the following subset of policies to study:

$$\begin{aligned} P &= \{P_d = \{d, d+1, ..., D\}, \text{ for } d = 1, 2, ..., D\} \\ &= \{\{1, 2, ..., D\}, \{2, 3, ..., D\}, ..., \{D-1, D\}, \{D\}\}. \end{aligned} \tag{4.3}$$

Since a policy P_d is defined by the value of d, the depth of its first iteration, we call d the "policy number". Recall that $P_D = P_7 = \{7\}$ is the degenerate case, a BFS of depth 7, currently used in the Gnutella network. Similarly, we looked only at a few possible W values: $\{1, 2, 4, 6, 150\}$. These values are measured in seconds.

In the first experiment, our client ran for a period of 4 days, submitting twice each query in our representative set Q_{rep}. Throughout this period, the client maintained 8 neighbors. When analyzing the data, we defined the desired number of results $Z = 50$. Later in this section, we will show what happens when Z is varied, and when the number of neighbors of our client changes.

We note that our client ran over an 10 Mb Ethernet connection. We wanted to ensure that the connection was stable (unlike dialup modems, and sometimes even cable modem and DSL), and we wanted to ensure that the connection did not saturate. Both of these characteristics were needed to ensure that our client would not die and interrupt the ongoing experiments. Most peer-to-peer clients will be connected via lower bandwidth connections, so we must keep in mind that the absolute numbers that we see in the following graphs may not be the same across all clients, though the tradeoffs should be comparable.

Cost Comparison. Figures 4.1 and 4.2 show the cost of each policy, for each value of W, in terms of average aggregate bandwidth and average aggregate processing cost, respectively. Along the x-axis, we vary d, the policy number. Immediately obvious in these figures are the cost savings. Policy P_1 at $W = 8$ uses just about 19% of the aggregate bandwidth per

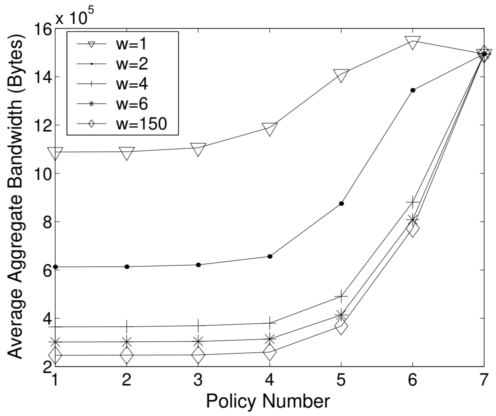

Figure 4.1: Bandwidth consumption for various Iterative Deepening policies

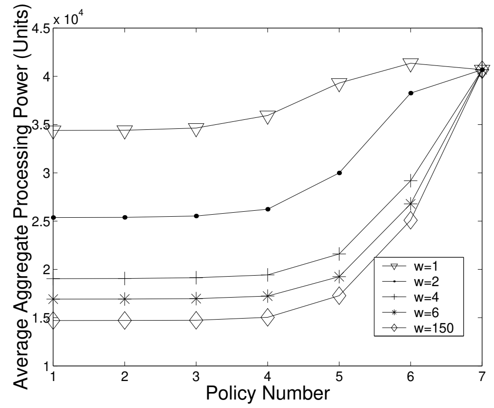

Figure 4.2: Processing cost for various Iterative Deepening policies

query used by the BFS technique, P_7, and just 40% of the aggregate processing cost per query.

To understand how such enormous savings are possible, we must understand the tradeoffs between the different policies and waiting periods. The curves in Figures 4.1 and 4.2 have roughly the same shape, for the same reasons. Let us therefore focus on bandwidth consumption, in Figure 4.1. First, notice that the average aggregate bandwidth for $d = 7$ is the same, regardless of W (also seen in Figure 4.3). Since the waiting time W only comes into play between iterations, it does not affect $P_7 = \{7\}$, which has only a single iteration.

Next, notice that as d increases, the bandwidth consumption of policy P_d increases as well. The larger d is, the more likely the policy will waste bandwidth by sending the query out to too many nodes. For example, if a query Q can be satisfied at a depth of 4, then policies P_5, P_6 and P_7 will "overshoot" the goal, sending the query out to more nodes than necessary to satisfy Q. Sending the query out to more nodes than necessary will generate extra bandwidth from forwarding the Query message, and transferring Response messages back to the source. Hence, as d increases, bandwidth consumption increases as well, giving P_7 the worst bandwidth performance.

Now, notice that as W decreases, bandwidth usage increases. If W is small, there is a higher likelihood that the client will prematurely determine that the query was not satisfied, leading to the "overshooting" effect we described for large policy numbers. For example, say $W = 6$ and $d = 4$. If a query Q can be satisfied at depth 4, but 8 seconds are required before Z results arrive at the client, then the client will only wait for 6 seconds, determine that the query is not satisfied, and initiate the next iteration at depth 5. In this case, the client overshoots the goal. The smaller W is, the more often the client will overshoot; hence, bandwidth usage increases as W decreases.

Note also that for $W = 1$, when d is large, bandwidth usage actually decreases as d increases. The reason for this anomaly is the cost of Resend messages. As we just explained, when W is small, the client tends to overshoot the goal. This effect is so strong for $W = 1$ that almost every query under P_6 will eventually reach depth 7. Therefore, roughly the same amount of bandwidth is used for Query and Response messages for P_6 and P_7, but policy P_6 also has the additional cost of Resend messages for almost every query. Aggregate bandwidth consumption for P_6 is therefore greater than that of P_7.

Quality of Results. Recall that one of the strengths of Iterative Deepening is that it can decrease the cost of queries without detracting from its ability to satisfy queries. Hence, satisfaction under any Iterative Deepening policy is equal to satisfaction under the current BFS scheme used by Gnutella, which, according to our data, equals .64. Because the Iterative Deepening technique is best applied in cases where satisfaction, and not the number of responses, is the more appropriate metric, we will not compare policies by the number of results returned per query.

The remaining quality of results metric, time to satisfaction, is shown in Figure 4.3 for each policy and value of W. We see that there is an inverse relationship between time to satisfaction and cost. As W increases, the time spent for each iteration grows longer. In addition, as d decreases, the number of iterations needed to satisfy a query will increase, on average. In both of these cases, the time to satisfaction will increase. Note, however, that delay is often caused by saturated connections or thrashing nodes. If all nodes in Gnutella were to use Iterative Deepening, load on nodes and connections would decrease considerably, thereby decreasing these delays. Time to satisfaction should therefore grow less quickly than is shown in Figure 4.3, as d increases or W increases.

In deciding which policy would be the best to use in practice, we must keep in mind what time to satisfaction the user can tolerate in an interactive system. Suppose a system requires the average time to satisfaction to be no more than 9 seconds. Looking at Figure 4.3, we see that several combinations of d and W result in this time to satisfy, e.g., $d = 4$ and $W = 4$, or $d = 5$ and $W = 6$. Looking at Figures 4.1 and 4.2, we see that the policy and waiting period that minimizes cost, while satisfying the time constraint, is P_5 and $W = 6$ (with

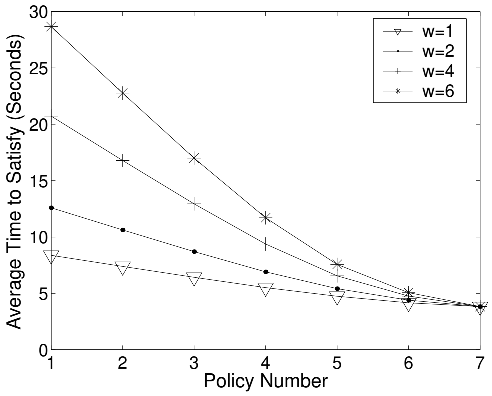

Figure 4.3: Time to satisfaction for various Iterative Deepening policies

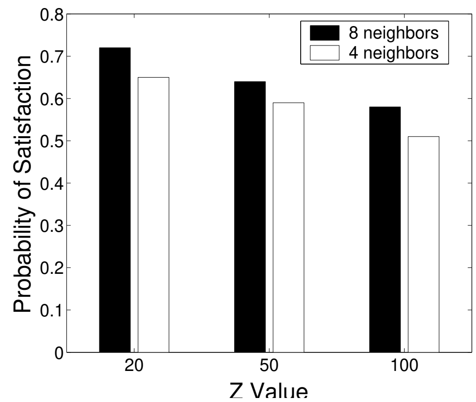

Figure 4.4: Probability of satisfaction for different Z

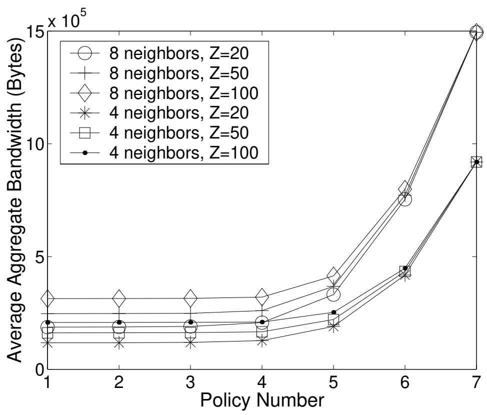

Figure 4.5: Bandwidth consumption for Iterative Deepening at $W = 150$

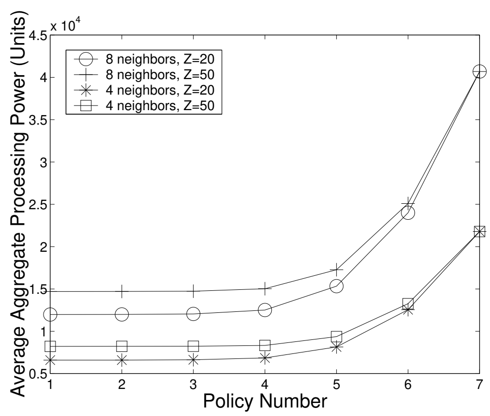

Figure 4.6: Processing cost for Iterative Deepening at $W = 150$

savings of 72% in aggregate bandwidth and 53% in aggregate processing cost over BFS). We would therefore recommend this policy for our system.

Variations. Performance of the Iterative Deepening technique and its various policies rely on the value chosen for Z. In addition, performance depends on the number of nodes that process each query, which in turn is determined locally by the number of neighbors (degree) that a client maintains. To see the effect these two factors have on the Iterative Deepening technique, we ran our data collection experiment twice, first where our client maintained 8 neighbors, then where our client maintained 4 neighbors. Over each data set, we then ran analysis three times, first with $Z = 20$, then with $Z = 50$, then with $Z = 100$.

Figure 4.4 shows the performance of each variation in terms of satisfaction. Since choice of policy number and W does not affect the satisfaction of Iterative Deepening, this figure represents all Iterative Deepening policies. As expected, when the definition of satisfying becomes more stringent (i.e., Z increases), satisfaction decreases. However, it is encouraging to note that satisfaction does not drop very quickly as Z increases. A value of $Z = 100$ requires that 5 times as many results need to be returned in order to satisfy the query, when compared to $Z = 20$. Yet, its probability of satisfying is only 15% less with 8 neighbors, and 14% less with 4 neighbors.

Also as expected, when the number of neighbors of our client decreases, the probability that a query is satisfied also decreases, since fewer neighbors generally translates to fewer number of results. However, we find it interesting that satisfaction with 4 neighbors is not much lower than satisfaction with 8 neighbors. Though we have half as many neighbors, our probability of satisfaction is only about 6% lower when we have four neighbors as opposed to 8. The reason for this effect is that with 8 neighbors, our client usually received far more results than was needed to satisfy the query. When the number of neighbors decreased to 4, the client received significantly fewer results, but in most cases it was still enough to satisfy the query.

The cost savings of having fewer neighbors, however, is quite significant. Figures 4.5 and 4.6 show the cost of the different variations for $W = 150$ in terms of average aggregate bandwidth and average aggregate processing cost, respectively. We chose $W = 150$ because it represents the best-case cost performance of these variations from our experimental data. Again along the x-axis we vary the policy number d. We see in Figure 4.5 that for all policy numbers, Iterative Deepening with 4 neighbors requires just 55% to 65% of the bandwidth required with 8 neighbors. Similarly, Figure 4.6 shows us that Iterative Deepening with 4 neighbors requires just 55% of the processing cost required with 8 neighbors. Though we do not show the cost of the policies for different values of W, we verify that the same magnitude of savings occur for all W we considered. Hence, decreasing the number of neighbors that the query is sent to might decrease satisfaction a little, but decrease the cost significantly. Larger values of Z translate to more costly queries, because more results must be processed and returned before the query terminates.

In terms of making a recommendation for a real system, we first note that most nodes do not have many neighbors, so the option of sending the query to fewer neighbors does not exist for them. For the nodes that do have many neighbors, they would be doing a service to the network by decreasing the number of neighbors they send the query to, if they are

Symbol	Heuristic: Select neighbor that...
RAND	(Random)
>RES	Returned the greatest number of results in the past 10 queries
<TIME	Had the shortest average time to satisfaction in the past 10 queries
<HOPS	Had the smallest average number of hops taken by results in the past 10 queries
>MSG	Sent our client the greatest number of messages (all types)
<QLEN	Had the shortest message queue
<LAT	Had the shortest latency
>DEG	Had the highest degree (number of neighbors)

Table 4.7: Heuristics used for Directed BFS

the query source. However, it may be difficult to convince these highly connected nodes to be altruistic, especially since they already provide a service to the network by routing much traffic.

4.5.2 Directed BFS

We studied 8 heuristics for Directed BFS, listed in Table 4.7. The RAND heuristic, choosing a neighbor at random, is used as the baseline for comparison with other heuristics.

Quality of Results. Figure 4.7 shows the probability of query satisfaction, for the different heuristics, for different values of Z. All heuristics except <HOPS have a marked improvement over the baseline heuristic RAND. In particular, >RES, sending the query to the neighbor that has produced the most results in past queries, has the best performance. It is followed by <TIME, sending the query to the neighbor that has produced results with the lowest time to satisfaction in past queries. We therefore see, as expected, that past performance is a good indicator of future performance. As with Iterative Deepening in Figure 4.4, increasing Z decreases the satisfaction. We could not find an intuitive explanation for why the performance of <QLEN drops at $Z = 100$.

Figure 4.8 shows the time to satisfaction of the different heuristics. Under this metric, again most heuristics outperform the baseline RAND. The <TIME heuristic has the best performance (36% lower than the baseline), followed by >RES (20% lower than the baseline). Again, we see that past performance is a good indicator of future performance.

We were surprised to see that >DEG, sending the query to the neighbor with the highest degree, did not perform as well as the other heuristics, with the exception of <HOPS. Past work such as [1] show that in a power-law network such as Gnutella, sending the query to nodes with the highest degree should allow one to reach many nodes in the network, and

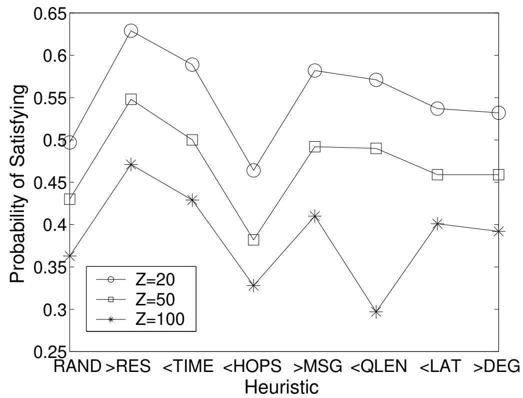

Figure 4.7: Probability of satisfaction for various Directed BFS policies

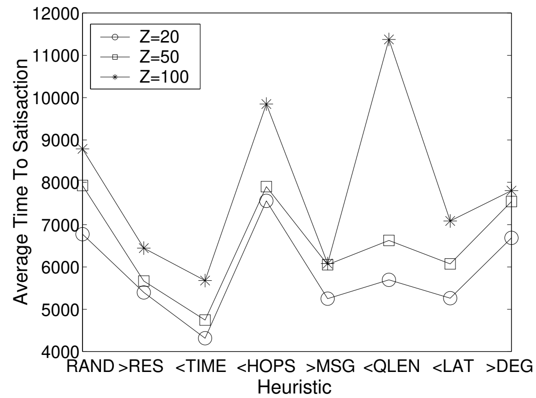

Figure 4.8: Time to satisfaction for various Directed BFS policies

therefore get many results back. However, this past work did not have real-time data to make decisions as we did, such as the past performance of neighbors, the amount of time a neighbor has been alive, etc. It is possible that with this additional real-time information, their routing algorithm might be enhanced.

We were also initially surprised to see that <HOPS, sending the query to the neighbor that has returned results with the lowest number of hops, performed so poorly in terms of both satisfaction and time to satisfaction. On closer examination of our logs, we found that a low average number of hops does not imply that the neighbor is close to other nodes with data, as we had hypothesized. Instead, it typically means that only a few nodes can be reached through that neighbor, but those few nodes are close to the neighbor. Though all results come from just a few hops away, only few results are returned through that neighbor. Since this heuristic returns few results, it also has the worst time to satisfaction. Though the average arrival time of results is lower with this heuristic than with others, the time at which the Zth result arrives for this heuristic is still higher than the time at which the Zth result arrives for other heuristics, especially if Z is large.

Cost. Figures 4.9 and 4.10 show the cost of Directed BFS under each heuristic, in terms of average aggregate bandwidth and average aggregate processing cost, respectively. The cost of Directed BFS in unaffected by the value of Z, so the single curve in these figures represent all values of Z. We see a definite correlation between cost and quality of results. Many of our heuristics return higher quality results than RAND because they select neighbors that are directly or indirectly connected to many other nodes. Because more nodes process

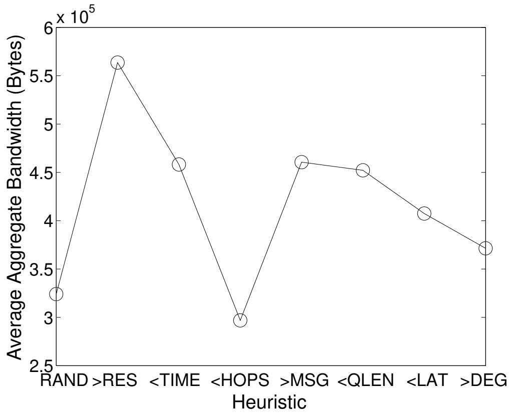

Figure 4.9: Bandwidth consumption for Directed BFS

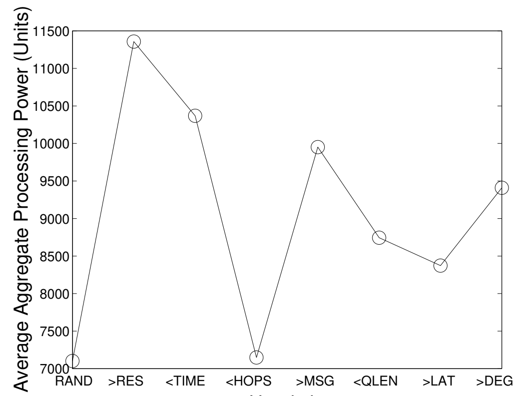

Figure 4.10: Processing cost for Directed BFS

the query when using these heuristics, more quality results are returned, but also more aggregate bandwidth and processing power is consumed.

We feel that since users of a system are more acutely aware of the quality of results that are returned, rather than the aggregate cost of a query, the heuristics that provide the highest quality results would be most widely accepted in open systems such as Gnutella. We would therefore recommend >RES or <TIME. Both heuristics provide good time to satisfaction, and a probability of satisfaction that is 9% and 13% lower than BFS with 8 neighbors, respectively. Furthermore, despite the fact that they are the most costly heuristics, they still require roughly 73% less processing cost than BFS, and 65% less bandwidth.

Compared with Iterative Deepening, the strength of Directed BFS is time to satisfaction. Comparing Figures 4.3 and 4.8, we see that Directed BFS heuristics yield times to satisfaction comparable to the best times achievable by Iterative Deepening. However, by sacrificing time to satisfaction, Iterative Deepening can achieve lower cost than any Directed BFS heuristic. Furthermore, satisfaction for all Iterative Deepening policies is higher than satisfaction of any Directed BFS heuristic. Table 4.9 in Section 4.6 summarizes the comparisons between these two techniques.

4.5.3 Local Indices

To evaluate the Local Indices technique, we used data collected from the Iterative Deepening experiments described in Section 4.4.1, where the client was run with 8 neighbors. For the sake of comparison, we evaluate only those policies that have the same performance as a

Policies
$P_0 = \{1, 2, 3, 4, 5, 6, 7\}$
$P_1 = \{0, 3, 6\}$
$P_2 = \{0, 5\}$
$P_3 = \{4\}$
$P_4 = \{3\}$
$P_5 = \{2\}$
$P_6 = \{1\}$
$P_7 = \{0\}$

Table 4.8: Policies for Local Indices

BFS of depth D, in terms of number of results and satisfaction. Hence we only consider policies where the last depth is set to $D - r$ (see Section 4.3.3 for an explanation). Since the number of nodes grows exponentially as distance from the source node increases, to minimize cost, we want the remaining depths in the policy to be as small as possible. If a query is being executed at depth x, then the previous depth in the policy need be at least $x - 2r - 1$ (processing at depth $x - 2r - 1$ covers all nodes up to depth $x - r - 1$, and processing at depth x covers all nodes starting at $x - r$). As a result, for every possible radius r, we evaluate a single policy P_r, defined as $P_r = \{..., D - 3r - 1, D - r\}$. For clarity, Table 4.8 lists these policies explicitly.

Cost. In the following cost evaluations, we vary `QueryJoinRatio` because cost is highly dependent on this parameter. The default value for `QueryJoinRatio` observed in Gnutella is roughly 10 [90]. Recall that `QueryJoinRatio` gives us the ratio of queries to joins/leaves in the network (i.e., roughly how many queries a user submits during a session). However, here we do not vary the value of `QueryUpdateRatio`, the ratio of queries to updates in the network, because we found that varying this parameter had the same types of effects as varying `QueryJoinRatio`. The default value of this parameter was obtained indirectly, because it was impossible to observe how frequently users updated their collections. We first assumed that most online updates come as a result of users downloading files from other users. We then use the ratio of queries to downloads observed in [94] for the OpenNap system as an estimate for Gnutella. Our estimate for `QueryUpdateRatio` is therefore 0.5, meaning that on average, a user updates 2 files for every query he submits.

Figures 4.11 and 4.12 give us the cost of the Local Indices policies for various values of `QueryJoinRatio` (QJR), in terms of average aggregate bandwidth and average aggregate processing cost, respectively. Along the x-axis, we vary the radius r. Recall that $r = 0$ is the

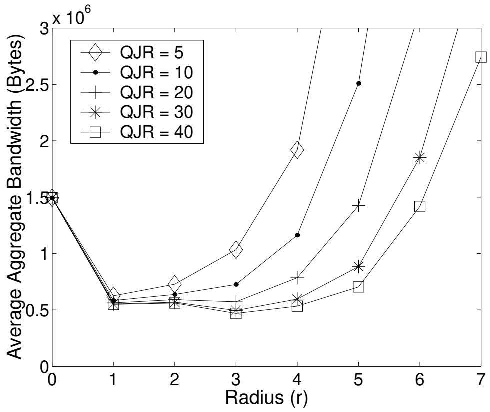

Figure 4.11: Bandwidth consumption for Local Indices

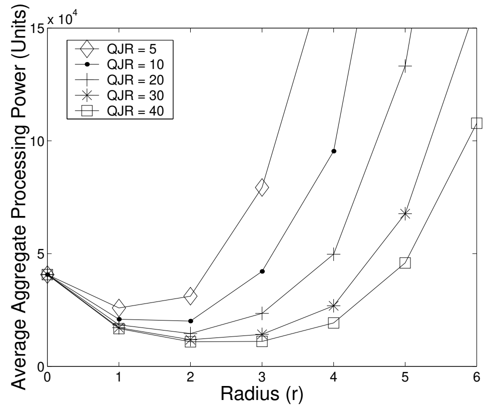

Figure 4.12: Processing cost for Local Indices

degenerate case where a node keeps an index over its own collection only; P_0 is equivalent to a BFS of depth D.

Again, we can see huge cost savings from the Local Indices technique, especially as `QueryJoinRatio` increases. At `QueryJoinRatio` $= 10$, policy P_1 uses about $5.8 \cdot 10^5$ bytes of aggregate bandwidth, which is about 39% of what the default BFS scheme uses (i.e., $r = 0$), and it uses $2.1 \cdot 10^4$ units of processing power, which is about 51% of what the default scheme uses. When `QueryJoinRatio` $= 100$, only 28% of the bandwidth and 21% of the processing cost of the default scheme is required. We see from both figures that cost decreases as `QueryJoinRatio` increases. Because there are more queries for every join and leave in the network as `QueryJoinRatio` increases, the amortized cost of joins and leaves decreases. However, the cost for $r = 0$ is the same for all values of `QueryJoinRatio` because it is the degenerate case where a node keeps an index over its own collection only. At $r = 0$, the cost of joins, leaves and updates is zero.

In Figures 4.11 and 4.12, as r increases, the cost of the policies decrease, and then increase again. To understand the reason for this behavior, we looked at the cost of queries, the amortized cost of joins and leaves, and the amortized cost of updates separately in Figure 4.13, for `QueryJoinRatio`=20. When r is small, the cost of queries is quite large, and dominates over the cost of joins, leaves and updates. As r increases, the cost of queries decreases, because fewer nodes need to process the query, and because messages need to travel fewer hops. However, the amortized cost of joins and leaves increases exponentially, such that when r is large, this cost dominates over the cost of queries. The point at which total cost is minimized is somewhere in between the two extremes – in the case of Figure 4.13,

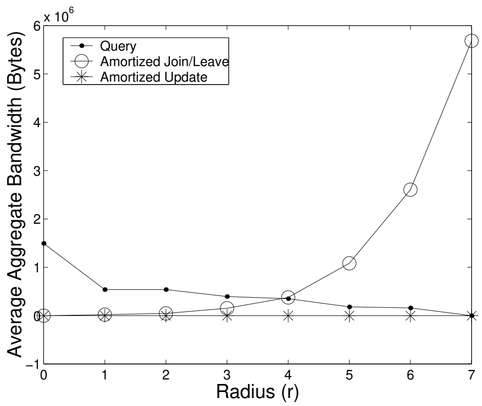

Figure 4.13: Comparison of Bandwidth Consumed by Queries and Join/Leaves

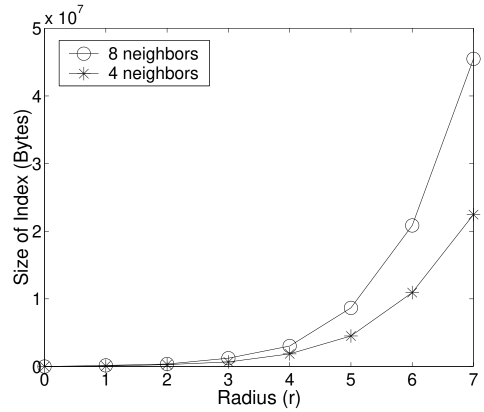

Figure 4.14: Size of the Index for different Radii(r)

it is at $r = 1$. The amortized cost of updates is always a relatively small fraction of total cost.

While in some cases, relatively large r have the best performance, we must keep in mind the storage requirements for the index at each node. Figure 4.14 shows us the size of the index that our node would have as a function of r, had we implemented Local Indices. The size requirement for a policy is simply the number of nodes within r hops, multiplied by the average number of files per user, and the average size of each file metadata. The number of neighbors directly affect the size of the index, so we show results for our client with 4 and 8 neighbors. Since our node was relatively well-connected (average degree is roughly 2 in Gnutella), the numbers shown in Figure 4.14 are significantly larger than they would be for an average node.

Though the size of the index grows exponentially, it is still practical for all r in our range. At $r = 7$ with 4 neighbors, the size of our index would have been roughly 21 MB, which is not unreasonable, but perhaps larger than many users would like to keep in memory for the sake of the application. For $r = 1$, the size of our index would have been roughly 71 KB – an index so small that almost certainly no users would object to keeping.

For today's system with `QueryJoinRatio` $= 10$, we recommend using $r = 1$, because it achieves the greatest savings in cost (61% in bandwidth, 49% in processing cost), and the index size for $r = 1$ is so small, it should not cause any barrier to implementation. In the future, when `QueryJoinRatio` increases, the best value for r will also increase.

Technique	Time to Satisfy	Probability of Satisfaction	Number of Results	Aggregate Bandwidth Cost	Aggregate Processing Cost
BFS	100%	100%	100%	100%	100%
Iterative Deep. ($d = 5$, $W = 6$)	190%	100%	19%	28%	47%
Directed BFS (>RES)	140%	86%	37%	38%	28%
Local Indices ($r = 1$)	$\approx$ 100%	100%	100%	39%	51%

Table 4.9: Relative performance of techniques, using BFS as the baseline. For each technique, we show the performance of a single policy we recommend for today's systems.

Time to Satisfaction. In the absence of data from a system that actually uses indices, calculating time to satisfaction for Local Indices is very hard. Thus, we only qualitatively discuss this metric for Local Indices when compared to BFS.

For the typical query, most results under a BFS with depth limit D will come from D hops away. In a Local Indices policy, most results will come from $D - r$ hops away. Because most Query and Response messages under Local Indices have r fewer hops to travel, the time to forward messages to the outermost depth and back to the source will be shorter (for $r > 0$) than for BFS. However, because nodes have larger indices, processing the query should take more time. It is difficult to tell how these effects balance each other out as r changes, but we believe that time to satisfaction changes *slowly* with r. Hence, time to satisfaction for Local Indices with small r is comparable to time to satisfaction for BFS (as BFS is equivalent to Local Indices with $r = 0$).

4.6 Conclusion

This chapter presented the design and evaluation of three efficient search techniques, over an unstructured, pure P2P system. Compared to techniques used in existing systems, these new techniques greatly reduce the aggregate cost of processing queries over the entire system, while maintaining equally high quality of results. Table 4.9 summarizes the performance tradeoffs we have presented among our proposed techniques, as compared to the BFS technique currently in use. Because of the simplicity of these techniques and their excellent performance, we believe they can make a large positive impact on both existing and future pure P2P systems.

Chapter 5

Designing a Super-Peer Network

As we have seen in the previous two chapters, there are several types of P2P systems that reflect varying degrees of centralization. In *pure* systems such as Gnutella [35] and Freenet [30], all peers have equal roles and responsibilities in all aspects: query, download, message routing, etc. In a *hybrid* system such as Napster [62], search is performed over a centralized directory, but download still occurs in a P2P fashion – hence, peers are equal in download only.

Super-peer networks such as KaZaA [51] (by far the most popular file-sharing system today) present a cross between pure and hybrid systems. A *super-peer* is a node that acts as a centralized server to a subset of clients. Clients submit queries to their super-peer and receive results from it, as in a hybrid system. However, super-peers are also connected to each other as peers in a pure system are, routing messages over this overlay network, and submitting and answering queries on behalf of their clients and themselves. Hence, super-peers are equal in terms of search, and all peers (including clients) are equal in terms of download. A "super-peer network" is simply a P2P network that consists of these super-peers and their clients.

Although P2P systems have many strengths, each type of system also has its own weaknesses. Pure P2P systems tend to be inefficient; for example, search in Gnutella consists of flooding the network with query messages. Much existing research, including that presented in the previous chapter, has focused on improving the search protocol in pure systems.

Another important source of inefficiency in pure systems is bottlenecks caused by the very limited capabilities of some peers. For example, the Gnutella network experienced deteriorated performance – e.g., slower response time, fewer available resources – when the size of the network surged in August 2000. One study [90] found these problems were caused by peers connected by dial-up modems becoming saturated by the increased load, dying and fragmenting the network by their departure. Peers on modems were dying because all peers in Gnutella are given equal roles and responsibilities, regardless of capability. However, studies such as [82] have shown considerable heterogeneity (e.g., up to 3 orders of magnitude difference in bandwidth) among the capabilities of participating peers. The obvious conclusion is that an efficient system should take advantage of this heterogeneity, assigning greater responsibility to those who are more capable of handling it.

Hybrid systems also have their shortcomings. While centralized search is generally more efficient than distributed search in terms of aggregate cost, the cost incurred on the single node housing the centralized index is very high. Unless the index is distributed across several nodes, this single node becomes a performance and scalability bottleneck. Hybrid systems are also more vulnerable to attack, as there are few highly-visible targets that would bring down the entire system if they failed.

Because a super-peer network combines elements of both pure and hybrid systems, it has the potential to combine the efficiency of a centralized search with the autonomy, load balancing and robustness to attacks provided by distributed search. For example, since super-peers act as centralized servers to their clients, they can handle queries more efficiently than each individual client could. However, since there are relatively many super-peers in a system, no single super-peer needs to handle a very large load, nor will one peer become a bottleneck or single point of failure for the entire system (though it may become a bottleneck for its clients, as described in Section 5.1).

For the reasons outlined above, super-peer networks clearly have potential; however, their design involves performance tradeoffs and questions that are currently not well understood. For example, what is a good ratio of clients to super-peers? Do super-peers actually make search more efficient (e.g., lower cost, faster response times), or do they simply make the system more stable? How much more work will super-peers handle compared to clients? Compared to peers in a system with no super-peers? How should super-peers connect to each other – can recommendations be made for the topology of the super-peer network? Since super-peers introduce a single-point of failure for its clients, are there ways to make them more reliable?

In this chapter, our goal is to develop practical guidelines on how to design super-peer networks, answering questions such as those presented above. In particular, our main contributions are:

- We present several "rules of thumb" that summarize the main tradeoffs in super-peer networks (Section 5.3.1).
- We formulate a procedure for the global design of a super-peer network, and illustrate how it improves the performance of existing systems (Section 5.3.2).
- We give guidelines for local decision-making at a super-peer to achieve a globally efficient network (Section 5.3.3).
- We introduce "k-redundancy", a new variant of super-peer design, and show that it improves both reliability and performance of the super-peer network.

By carefully studying super-peer networks and presenting our results here, our goal is to provide a better understanding of these networks that can lead to improved systems.

The remainder of the chapter is organized as follows. Section 5.1 gives a formal description of the search problem in super-peer networks and outlines the different parameters describing a super-peer network. Section 5.2 describes the framework for analysis used to generate our results, and Section 5.3 presents these results in the form of guidelines.

Many contributions in this chapter were published initially in [96].

5.1 Problem Description

A super-peer network operates exactly like a pure P2P network, as described in Section 4.2, except that every "node" in the previous description is actually a super-peer, and each super-peer is connected to a set of clients. Clients are connected to a single super-peer only. Figure 5.1a illustrates what the topology of a super-peer network might look like. We call a super-peer and its clients a *cluster*, where *cluster size* is the number of nodes in the cluster, including the super-peer itself. A pure P2P network is actually a "degenerate" super-peer network where cluster size is 1 – every node is a super-peer with no clients.

When a super-peer receives a query from a neighbor, it will process the query on its clients' behalf, rather than forwarding the query to its clients. In order to process the query for its clients, a super-peer keeps an index over its clients' data. This index must hold

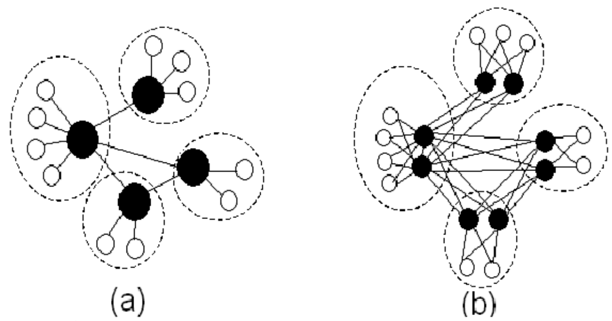

Figure 5.1: Illustration of a super-peer network (a) with no redundancy, (b) with 2-redundancy. Black nodes represent super-peers, white nodes represent clients. Clusters are marked by the dashed lines.

sufficient information to answer all queries. For example, if the shared data are files and queries are keyword searches over the file title, then the super-peer may keep inverted lists over the titles of files owned by its clients. If the super-peer finds any results, it will produce one Response message. This Response message contains the results, and the address of each client whose collection produced a result.

In order for the super-peer to maintain this index, when a client joins a cluster, it will send metadata over its collection to its super-peer, and the super-peer will add this metadata to its index. When a client leaves, its super-peer will remove its metadata from the index. If a client ever updates its data (e.g., insertion, deletion or modification of an item), it will send this update to the super-peer as well. Hence, in super-peer networks, joins (and the associated leave) and updates are much more complex and costly than they are in a pure network. Joins, leaves and updates are handled in a fashion similar to Local Indices described in Chapter 4, except now only super-peers maintain indices.

When a client wishes to submit a query to the network, it will send the query to its super-peer only. The super-peer will then submit the query to its neighbors as if it were its own query, and forward any Response messages it receives back to the client. Outside of the cluster, a client's query is indistinguishable from a super-peer's query.

Since clients are shielded from all query processing and traffic, weak peers can be made into clients, while the core of the system can run efficiently on a network of powerful super-peers. Hence, as mentioned earlier, super-peer networks use the heterogeneity of peers to their advantage. Also, as we will see in Section 5.3, the overhead of maintaining an index at the super-peer is small in comparison to the savings in query cost this centralized index allows.

We note that any message routing protocol for pure networks may be used on top of a super-peer network. For the purposes of this chapter, we employ the baseline Gnutella search (Section 4.2), where queries are flooded with a given TTL. Under this protocol, we define the *reach* of a query to be the number of nodes that process the query (super-peers), or have the query processed on their behalf (clients).

Super-peer redundancy. Although clusters are efficient, a super-peer becomes a single point of failure for its cluster, and a potential bottleneck. When the super-peer fails or simply leaves, all its clients become temporarily disconnected until they can find a new super-peer to connect to.

To provide reliability to the cluster and decrease the load on the super-peer, we introduce redundancy into the design of the super-peer. We say that a super-peer is *k-redundant* if there are k nodes sharing the super-peer load, forming a single "virtual" super-peer. Every node in the virtual super-peer is a *partner* with equal responsibilities: each partner is connected to every client and has a full index of the clients' data, as well as the data of other partners. Clients send queries to each partner in a round-robin fashion; similarly, incoming queries from neighbors are distributed across partners equally. Hence, the query load on each partner is a factor of k less than on a single super-peer with no redundancy.

A k-redundant super-peer has much greater availability and reliability than a single super-peer. Since all partners can respond to queries, if one partner fails, the others may continue to service clients and neighbors until a new partner can be found. The probability that all partners will fail before any failed partner can be replaced is much lower than the probability of a single super-peer failing.

However, super-peer redundancy comes at a cost. In order for each partner to have a full index with which to answer queries, a client must send metadata to each of these partners when it joins. Hence, the aggregate cost of a client join action is k times greater than before. Also, neighbors must be connected to each one of the partners, so that any partner may receive messages from any neighbor. Assuming that every super-peer in the network is k-redundant, the number of open connections increases by a factor of k^2. Because the number of open connections increases so quickly as k increases, in this chapter we will only consider the case where $k = 2$. Henceforth, we will use the term "super-peer redundancy" to refer to the 2-redundant case only. Figure 5.1b illustrates a super-peer network topology with redundancy.

In terms of eliminating bottlenecks and reducing load, one may wonder whether a more effective method than redundancy would be to simply make each partner into a super-peer with half the clients – that is, have twice the number of clusters at half the original size and no redundancy. In this way, the individual query load on each super-peer will be halved as with 2-redundancy, and the index would be half the size. Aside from the obvious loss of reliability with this new method, splitting the cluster in two actually has the surprising effect of putting a *greater* load on each super-peer than in the 2-redundant case, a result we will see in Section 5.3.1.

Topology of the super-peer network. Gnutella is the only open P2P system for which topology information is known. In Gnutella, the overlay network formed by the peers follows a power-law, meaning the frequency f_d of an outdegree d is proportional to d^{α}, where α is some constant. The power-law naturally occurs because altruistic and powerful peers voluntarily accept a greater number of neighbors. (We will see in Section 5.3.1 how a greater outdegree results in greater load).

From crawls of the Gnutella network performed in June 2001, we found the average outdegree of the network to be 3.1. In a super-peer network, however, we believe the average outdegree will be much higher, since super-peers have greater load capacity than an average peer. Because it is difficult to predict the average outdegree of a super-peer network, we will assume that every super-peer will be given a "suggested" outdegree from some global source (e.g., as part of the protocol). We assume the actual outdegrees will vary according to a power-law with this "suggested" outdegree as the average, since some super-peers will be more able and willing to accept a large outdegree than others.

5.2 Evaluation Model

We will compare the performance of super-peer networks in a file-sharing application using the metrics defined in Section 2.5. We will measure both aggregate and individual load in terms of incoming bandwidth, outgoing bandwidth and processing power. We will also measure quality of results by number of results. While our performance model does not directly capture response time, we will show in Section 5.3.1 how relative response times can be deduced by other means.

Name	Default	Description
Graph Type	Power	The type of network, which may be strongly connected or power-law
Graph Size	10000	The number of peers in the network
Cluster Size	10	The number of nodes per cluster
Redundancy	No	A boolean value specifying whether super-peer redundancy is used
Avg. Outdegree	3.1	The average outdegree of a super-peer
TTL	7	The time-to-live of a query message
Query Rate	$9.26 \cdot 10^{-3}$	The expected number of queries per user per second
Update Rate	$1.85 \cdot 10^{-2}$	The expected number of updates per user per second

Table 5.1: Configuration parameters and default values

5.2.1 Performance Evaluation

We will be comparing the performance of different *configurations* of systems, where a configuration is defined by a set of parameters, listed in Table 5.1. Configuration parameters describe both the topology of the network, as well as user behavior. We will describe these parameters in further detail as they appear later in the section.

There are 4 steps in the analysis of a configuration:

Step 1: Generating an instance. The configuration parameters listed in Table 5.1 describe the desired topology of the network. First, we calculate the number of clusters as $n = \frac{GraphSize}{ClusterSize}$. We then generate a topology of n nodes based on the *type* of graph specified. We consider two types of networks: *strongly connected* and *power-law*. We study strongly connected networks as a best-case scenario for the number of results (reach covers every node, so all possible results will be returned), and for bandwidth efficiency (no Response messages will be forwarded, so bandwidth is conserved). We study power-law networks because they reflect the real topology of the Gnutella network. Strongly connected networks are straightforward to generate. Power-law networks are generated according to the $PLOD$ algorithm presented in [71].

Each node in the generated graph corresponds to a single cluster. We transform each node into a single super-peer or "virtual" super-peer if there is redundancy. We then add C clients to each super-peer, where C follows the normal distribution $N(\mu_c, .2\mu_c)$, and where μ_c is the mean cluster size defined as $\mu_c = ClusterSize - 1$ if there is no redundancy and $\mu_c = ClusterSize - 2$ if there is. To each peer in the network, both super-peers and clients, we assign a number of files and a lifespan according to the distribution of files and lifespans measured by [82] over Gnutella.

Step 2: Calculating expected cost of actions. There are three "macro" actions in our cost model: query, join and update. Each of these actions is composed of smaller "atomic" actions for which costs are calculated in Section 2.4, in terms of bandwidth and processing resources.

As an example of how to calculate the cost of a "macro" action, consider the cost of a client joining the system. From the client perspective, the action consists of the startup cost of sending a Join message, and for each file in its collection, the cost of sending the metadata for that file to the super-peer. Suppose the client has x files and m open connections. Outgoing bandwidth for the client is therefore $80 + 72 \cdot x$, incoming bandwidth is 0, and processing cost is $.44 + .2 \cdot x + .01 \cdot m$. From the perspective of the super-peer, the client join action consists of the startup cost of receiving and processing the Join message, and then for each file owned by the client, the cost of receiving the metadata and adding the metadata to its index. In this example, we see how an action can involve multiple nodes, and how cost is dependent on the instance of the system (e.g., how many files the client owns). Updates, like joins, are a straightforward interaction between a client and super-peer, or just the super-peer itself.

Queries are much more complicated since they involve a large number of nodes, and depend heavily on the topology of the network instance. Again, for our evaluations, we assume the use of the simple baseline search used in Gnutella (described in Section 4.2); other protocols such as those described in Chapter 4 may also be used on a super-peer network, resulting in overall performance gain, but similar tradeoffs between configurations. We assume a message takes an equal amount of time to travel across any edge, and that each super-peer takes an equal amount of time to process and forward queries. Given these assumptions, to calculate the cost of a query, we may use a breadth-first traversal over the network to determine which nodes receive the query, where the source of the traversal is the query source S, and the depth is equal to the TTL of the query message. Any response message will then travel along the reverse path of the query, meaning it will travel up the predecessor graph of the breadth-first traversal until it reaches the source S. Every node along this path must sustain the cost of forwarding this message. In Section 5.2.2 we discuss how to determine the number of responses received for a query.

Step 3: Calculating load from actions. In step 2 we calculate expected values for C_{qST}, C_{jST} and C_{uST}, the cost of a query, join and update action, respectively, when the action is initiated by node S and incurred on node T, for every node S and and every node T in the network instance. S and T may be super-peers or clients.

For each type of action, we need to know the rate at which the action occurs. Default values for these rates are provided in Table 5.1. Query rate is taken from the general statistics listed in Table 2.1. Join rate is determined on a per-node basis. In our model, we make the simplifying assumption that when a node leaves the system, it is immediately replaced by a node joining the system. Hence, the rate at which nodes join the system is the inverse of the length of time they remain logged in. Update rate is obtained indirectly, since it is impossible to observe through experiments how frequently users updated their collections. We first assume that most online updates occur as a result of a user downloading a file. We then use the rate of downloads observed in Section 2.2 for the OpenNap system as our update rate. Because the cost of updates is low relative to the cost of queries and joins, the overall performance of the system is not sensitive to the value of the update rate.

Given the cost and rate of each type of action, we can now calculate the expected load on an individual node T, for a network instance I:

$$\begin{aligned} E[M_T|I] = \sum_{S \in \text{network}} & E[C_{qST}|I] \cdot E[F_{qS}] + \\ & E[C_{jST}|I] \cdot E[F_{jS}|I] + E[C_{uST}|I] \cdot E[F_{uS}] \end{aligned} \tag{5.1}$$

F_{qS} is defined as the number of queries submitted by S in a second, so $E[F_{qS}]$ is simply the query rate per user listed in Table 5.1, for all S. $F_{jS}|I$ and F_{uS} are similarly defined.

We can also calculate the expected number of results per query originated by node S:

$$E[R_S|I] = \sum_{T \in \text{network}} E[N_T|I] \tag{5.2}$$

Often we will want to calculate the expected load or results per query on nodes that belong to some set Q defined by a condition: for example, Q may be the set of all nodes that are super-peers, or the set of all super-peers with 2 neighbors. The expected load M_Q of all such nodes is defined as:

$$E[M_Q|I] = \frac{\sum_{n \in Q} E[M_n|I]}{|Q|} \tag{5.3}$$

Similarly, aggregate load is defined as:

$$E[\overline{M}|I] = \sum_{n \in \text{network}} E[M_n|I] \tag{5.4}$$

Step 4: Repeated Trials. We run analysis over several instances of a configuration and average $E[M|I]$ over these trials to calculate $E[E[M|I]] = E[M]$, the value by which we compare different configurations. We also calculate 95% confidence intervals for $E[M|I]$.

5.2.2 Query Model

To determine how many results a super-peer T returns, we use the query model developed in Section 2.3. Recall that this query model defines two probability functions: $g(i)$, which gives us the probability that a random submitted query is equal to query q_i, and $f(i)$, which gives us the probability that a random file will be a match (a result) for query q_i. We will use the same distributions experimentally determined for the OpenNap system in Section 2.3 for our default configuration, since OpenNap is also a file-sharing P2P system, and there is no available data from live super-peer networks.

Let x_i be the number of files owned by client i of super-peer T, and let T have c clients, and own x_T files. The total number of files indexed by T is $x_{tot} = x_T + \sum_{i=1}^{c} x_i$. The query model assumes that the probability that a given file will be returned as a match is independent of whether other files in the same collection are returned. With this assumption, the number of files returned from a collection of size n for a query of selection power p is distributed according to `binomial(n,p)`, and the expected value is therefore $n \cdot p$. Hence,

$$E[N_T|I] = \sum_{j=0}^{\infty} g(j) \cdot (f(j) \cdot x_{tot}) \tag{5.5}$$

Now, let Q_i be the identifier variable where $Q_i = 1$ iff client i returns at least one result. $P(Q_i|I) = 1 - P(\text{client } i \text{ returns no results}) = 1 - (1-p)^{x_i}$, where p is the selection power of the current query. Hence,

$$E[K_T|I] = \sum_{i=1}^{c} E(Q_i|I) = \sum_{i=1}^{c} \sum_{j=0}^{\infty} g(j) \cdot \left(1 - (1 - f(j))^{x_i}\right) \tag{5.6}$$

5.3 Results

In this section we present the results of our evaluations on a wide range of configurations. Because there are many different scenarios and factors to consider, we do not attempt to report on all the results here. Instead, from all our results we distill a few important "rules of thumb" to follow while designing a P2P topology, and present these rules to the reader, supported with examples from our experiments. We then formulate a general procedure that incorporates the rules and produces an efficient topology. Finally, we discuss how an individual node without a global view of the system might make local decisions to form a globally efficient network.

Recall that all results in the following section are expected values. All figures show the expected value of costs, along with vertical bars to denote 95% confidence intervals for $E[\text{value}|instance]$ where appropriate. All figures use the default parameters listed in Table 5.1, unless otherwise specified.

5.3.1 Rules of Thumb

The four rules of thumb we gathered from our experiments are as follows:

1. Increasing cluster size decreases aggregate load, but increases individual load.
2. Super-peer redundancy is good.
3. Maximize outdegree of super-peers.
4. Minimize TTL.

Let us now examine the causes, implications and details of each.

#1 Increasing cluster size decreases aggregate load, but increases individual load. In terms of cluster size, there is a clear tradeoff between aggregate and individual load. Figure 5.2 shows the aggregate bandwidth required by two systems as cluster size is varied. One system has a strongly connected topology with TTL=1, shown as a best-case scenario for bandwidth. The other system has a power-law topology with average outdegree of 3.1 and TTL=7, parameters reflective of the Gnutella topology. For now, ignore the curves for super-peer redundancy. In both systems described, the expected number of results is the same for all cluster sizes.

Both curves in Figure 5.2 show that aggregate load decreases dramatically at first as cluster size increases. Aggregate load then experiences a "knee" around cluster size of 200 in the strong network and 1000 in the power-law network, after which it decreases gradually. Intuitively, the fewer the super-peers in a system (i.e., the larger the cluster size), the less communication overhead between super-peers there will be. At the extreme where there is

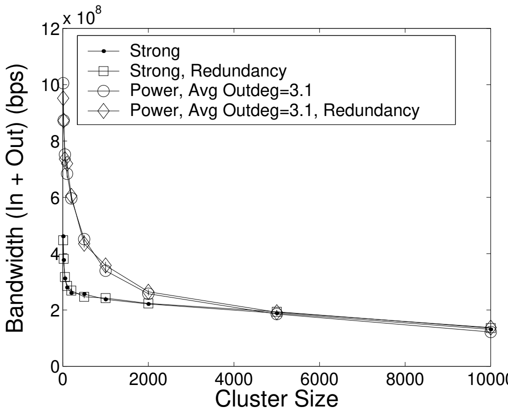

Figure 5.2: Aggregate load decreases with cluster size

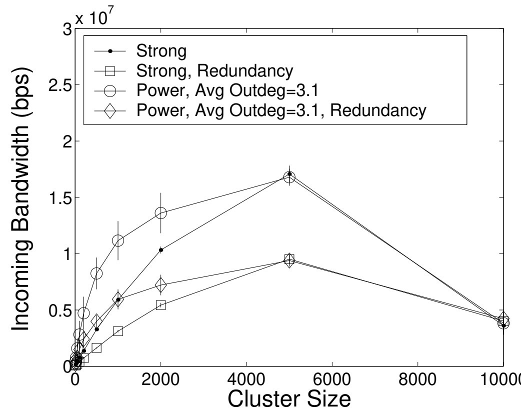

Figure 5.3: Individual load increases with cluster size, with few exceptions

a single super-peer (e.g., cluster size is 10000 in Figure 5.2), queries are sent directly from clients to a single "server", and results are sent directly from server to client. At the other extreme (e.g., cluster size is 1), the same number of results are being returned as in the first case, but in addition, there is the cost of sending queries to every super-peer, a startup cost for every super-peer as they process the query, and the overhead of additional packet headers for individual query responses. In fact, it is these extra costs, which grow inversely proportionally to the number of super-peers in the system, that cause the knee to occur in Figure 5.2.

Though large cluster size is ideal for aggregate load, it has the opposite effect on individual super-peer load, with a few exceptions. Figure 5.3 shows the individual incoming bandwidth required of super-peers for the same two systems as before, as cluster size is varied. With one exception, we see that individual load grows rapidly with the growth of cluster size. For example, in the strongly connected system, a super-peer with 100 clients has almost twice the load as a super-peer with 50.

The exception we see in this figure, occurring when there is a single super-peer (cluster size is 10000), only occurs for incoming bandwidth, and not for outgoing bandwidth or processing load. The explanation for this exception is as follows: Most incoming bandwidth load for a super-peer arises from forwarding query results from other super-peers to a client. If a cluster consists of a fraction f of all nodes in the network (i.e., $f = \frac{\text{cluster size}}{\text{number users}}$ in our example), then the super-peer for that cluster will be submitting roughly f of all queries on behalf of its clients. Because files in its own index constitute roughly f of all responses

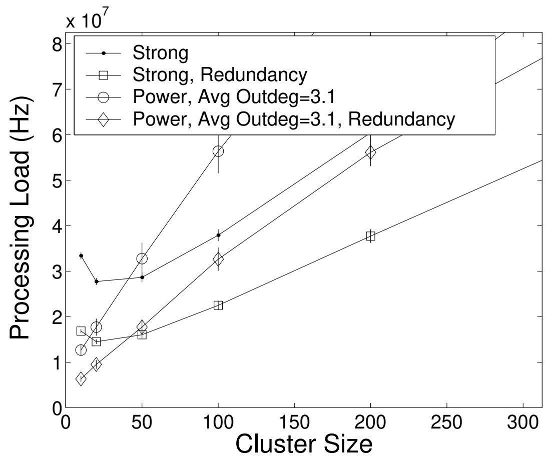

Figure 5.4: Individual processing load

to a query, $1 - f$ of all results will be sent from other clusters to this super-peer. Therefore, total expected incoming results is proportional to $f \cdot (1 - f)$, which has a maximum at $f = \frac{1}{2}$ (i.e., at a cluster size of 5000 in Figure 5.3), and two minimums at $f = 0$ and $f = 1$. Incoming bandwidth of a super-peer when cluster size is 10000 is thus much lower than when cluster size is 5000. This exception is important to keep in mind if incoming (downstream) bandwidth is the bottleneck, and we are considering large cluster sizes. Thus, a system designer should be careful to avoid cluster sizes around a tenth to a half of the total number of peers (i.e., where load exceeds load at $ClusterSize = GraphSize$).

Another exception to the rule can be found looking at processing load when outdegree is large. Figure 5.4 shows individual processing load as a function of cluster size, for the same two systems as before. Note that the processing load of a super-peer in the strongly connected graph actually increases as cluster size becomes very small. The reason for this behavior is the overhead of handling network traffic when the number of connections is large (see Appendix A for a discussion of this overhead). In a strongly connected graph, the number of connections is equal to the cluster size plus $\frac{\text{number users}}{\text{cluster size}}$, which reaches a maximum at very large and very small cluster sizes. The overhead of many open connections is high when cluster size is large, but relative to the cost of other actions (such as query processing), it is small. On the other hand, when cluster size is small, the majority of the processing load is due to this overhead. In most operating systems, the default number of open connections is limited, meaning overhead issues aside, having 1000+ open connections is not feasible. However, in the cases where it is feasible, this network overhead is an important consideration when selecting small cluster sizes. We will see how the number of open connections becomes a design consideration in Section 5.3.2.

What is a good tradeoff between aggregate and individual load? The answer to this question is specific to the application and situation. In general, individual good is probably more important than overall good, since users are less likely to altruistically be a super-peer if cost is very high. However, if all users belong to a single organization – for example, they are running on a corporate LAN – then aggregate load becomes quite important, as all costs are ultimately incurred on a single entity. In addition, a very important factor to consider is the availability and reliability of the system. With large cluster sizes and few super-peers, there are a few points of failure, which has a worse effect on availability should these super-peers leave, fail or be attacked. By keeping cluster sizes small, there are no easy targets for malicious users, and failure of a super-peer leaves just a few clients temporarily unconnected. In general, then, it would be good to choose a cluster size that is small enough to keep a reasonable individual load and provide reliability to the system, but large enough to avoid the knee in aggregate load when cluster size is small.

#2 Super-peer redundancy is good. Going back to Figure 5.2 and looking at the curves for super-peer redundancy, we see that introducing redundancy has no significant affect on aggregate bandwidth for both systems. In addition, Figure 5.3 shows that redundancy does decrease individual super-peer load significantly. For example, when cluster size is 100 in the strongly connected system, super-peer redundancy increases aggregate load by about 2.5%, but decreases individual load in each partner by 48% – driving it down to the individual load of a non-redundant super-peer when cluster size 40. Therefore, in terms of bandwidth, super-peer redundancy gives us better performance than both a system with no redundancy and the same cluster size, and a system with no redundancy and half the cluster size. Super-peer redundancy gives us the "best of both worlds" – the good aggregate load of a large cluster size, and the good individual load of a smaller cluster size.

However, super-peer redundancy involves a tradeoff between individual and aggregate load in terms of processing power. In the strongly connected system discussed earlier where cluster size is 100, aggregate processing load increases by roughly 17% when super-peer redundancy is introduced, while individual processing load decreases by about 41%. (Note that since the number of super-peers/partners doubles, aggregate load can increase though individual load decreases). Therefore, in a system such as Gnutella, super-peer redundancy is definitely recommended when individual load is the bottleneck, but should be applied

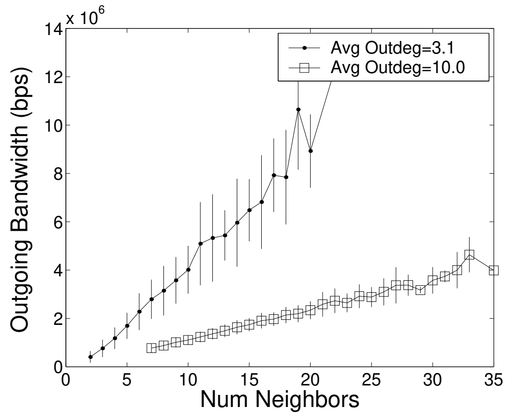

Figure 5.5: When all super-peers increase outdegree, individual and aggregate load decreases

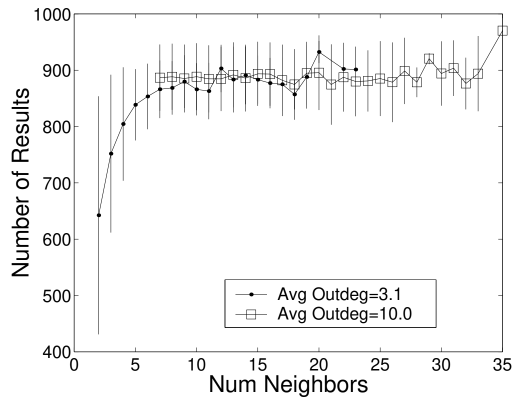

Figure 5.6: With a low average outdegree, many super-peers do not receive the full number of results

with care if aggregate processing load is a concern.

Later, we will also discuss the benefits of super-peer redundancy under different configurations.

#3 Maximize outdegree of super-peers. In the current Gnutella network, weak peers with few resources will connect to few neighbors, while altruistic powerful peers connect to many neighbors. This arrangement seems reasonable, since weak peers should not take on more load than they can handle. However, our experiments clearly show that increasing outdegree can actually *reduce* individual load, provided all peers do so. Figure 5.5 shows a histogram of individual incoming bandwidth loads for super-peers as a function of outdegree for two systems with power-law topologies: one where average outdegree is 3.1 (reflective of the Gnutella topology), and one where average outdegree is 10. Both systems have a cluster size of 20, though these results are the same for any cluster size. Figure 5.6 is like Figure 5.5, except it shows a histogram of expected number of results. As these figures present histograms, vertical bars denote one standard deviation, rather than confidence intervals.

First, note that while super-peers with very few (2 or 3) neighbors in the first topology do have a slightly lighter load than any super-peers in the second topology, they also receive fewer results. For example, in Figure 5.5, when average outdegree equals 3.1, the expected load of a super-peer with 3 neighbors is roughly $7.6 \cdot 10^5$ bps, while Figure 5.6 shows that this node receives roughly 750 expected results per query. In contrast, when average outdegree

Avg Outdegree	Incoming Bandwidth (bps)	Outgoing Bandwidth (bps)	Processing Power (Hz)
3.1	$3.51 \cdot 10^8$	$3.49 \cdot 10^8$	$6.06 \cdot 10^9$
10.0	$2.67 \cdot 10^8$	$2.65 \cdot 10^8$	$6.05 \cdot 10^9$

Table 5.2: Aggregate load comparison between an average outdegree of 3.1, and an average outdegree of 10.0

equals 10, the expected load of a super-peer with 7 neighbors is slightly higher, at $7.7 \cdot 10^5$ bps, but the expected number of results it receives is much higher, at roughly 890. Hence, the "gain" in load comes with a cost in user satisfaction. Once a super-peer in the first topology has enough neighbors (7) to attain full results, it has a load that is higher than that of most peers in the second topology! Super-peers in a system with higher outdegree therefore have a much lower load for a given number of results.

We can also see that super-peers with the highest outdegree in the first topology must support an extremely heavy load, while the loads of all peers in the second topology remain in the same moderate range. In this respect, the second topology can be said to be more "fair." Furthermore, fewer concentrated points of heavy load in the first topology translate to fewer points of failure, and therefore less reliability.

In terms of aggregate load, Table 5.2 shows an improvement of over 31% in bandwidth and a slightly lower processing load is attained by increasing outdegree from 3.1 to 10. In this table, we note that both topologies are assigned a cluster size of 100, since the first topology yields a smaller expected number of results for any smaller cluster size.

Increasing average outdegree improves performance because it reduces the *expected path length* of query responses. The expected path length (EPL) is simply the expected number of hops taken by a query response message on its path back to the query source. Each hop taken consumes incoming bandwidth from the receiver, outgoing bandwidth from the sender, and processing power from both; hence, the shorter the EPL, the more efficient a query is. For the systems shown in Figure 5.5, increasing average outdegree from 3.1 to 10 decreases EPL from 5.4 to 3.

For these benefits to apply, however, *all* super-peers must agree to increase their outdegree. Consider the system where average outdegree is 3.1 in Figure 5.5. If a single peer with 4 neighbors decides to increase its outdegree to 9, its load will increase from $1.18 \cdot 10^6$ to $3.58 \cdot 10^6$ bps, a 303% increase. If all super-peers increased their outdegree such that average outdegree is 10, then that super-peer will actually experience a 14% reduction in

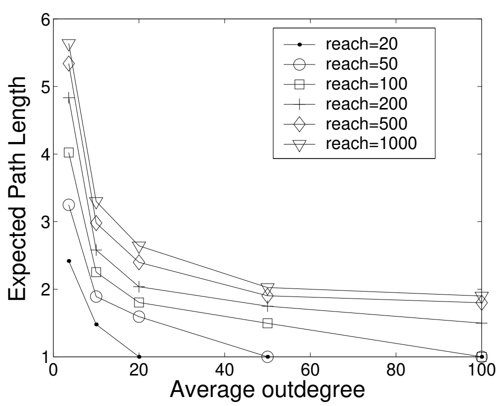

Figure 5.7: Estimating expected path length given reach and outdegree

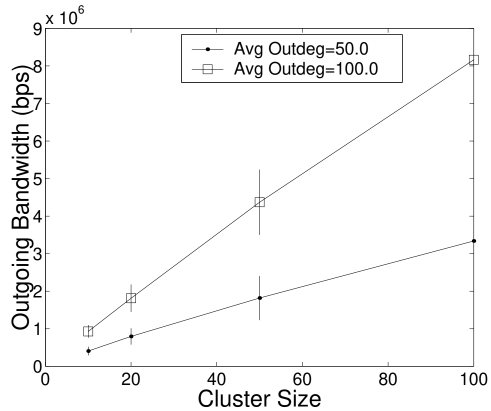

Figure 5.8: A modification to Rule #3: higher outdegrees can be harmful

load, to $1.01 \cdot 10^6$ bps.

In summary, if every super-peer agrees to increase outdegree, every super-peer's load will either decrease dramatically, or increase slightly accompanied by an increase in number of results. If only a few super-peers increase their outdegree, however, those few nodes will suffer a tremendous increase in load. It is important, therefore, that increasing outdegree be a uniform decision.

As a caveat to this rule in extreme cases, it is possible for the outdegree to be *too* large, in which case performance is negatively affected. To illustrate this point, Figure 5.7 shows the experimentally-determined EPL for a number of scenarios. Along the x-axis we vary the average outdegree of the topology (assumed to be power-law), and the different curves show different desired reaches. For example, when average outdegree is 20 and the desired reach is 500 nodes, Figure 5.7 tells us the EPL is roughly 2.5. We see in Figure 5.7 that in some cases, increasing outdegree does not significantly affect EPL. For example, when desired reach is 500 nodes, increasing outdegree from 50 to 100 only decreases EPL by .14. In these cases, having a larger outdegree will not only be useless in decreasing EPL, it will actually *hurt* performance by increasing the number of redundant queries. Figure 5.8 illustrates this point, showing the individual load of a super-peer with various cluster sizes, for an average outdegree of 50 and 100, and TTL=2. In each case, the desired reach is set to be the total number of super-peers, which is $\frac{\text{number users}}{\text{cluster size}}$. For all cluster sizes shown, systems with an average outdegree of 50 outperform systems with an average outdegree of 100 because the EPL is roughly the same in both cases, while redundant queries increase with outdegree. Therefore, as a modification to rule #3, outdegree should be increased, so long as doing so decreases EPL. Once EPL ceases to decrease, outdegree should not be increased any further.

#4 Minimize TTL. For a given outdegree and topology, reach is a function of TTL. Since the expected number of results is proportional to expected reach, a designer may choose the *desired reach* based on the desired number of results, and then select the TTL that will produce this reach. We will discuss how to select the TTL later in the section.

However, if the desired reach covers every super-peer in the network, then an infinite number of TTLs produce the correct reach, since if TTL=x allows all nodes to be reached, TTL=$x+1$ will have the same reach. In such a case, it is important to determine the minimum TTL needed to reach all nodes, and use that TTL. For example, a system with average outdegree=20 and TTL=4 uses roughly $7.75 \cdot 10^8$ bps in aggregate incoming bandwidth for a full reach. However, that same system uses only $6.30 \cdot 10^8$ bps, or 19% less aggregate incoming bandwidth, if TTL is reduced to 3 (which still allows a full reach). Similar improvements are seen for outgoing bandwidth and processing load.

The cause of this difference is the cost of sending unnecessary query messages when TTL > 3. Once queries have reached every node, any additional query message will be *redundant* – that is, it will have traveled over a cycle in the network, and arrive at a node that has already seen the query before. Any redundant query will be dropped by the receiver. Although query messages are not large, if every node sends a redundant query to each of its neighbors, the resources consumed can add up to a significant amount.

When reach covers every node in the system, the simplest way to choose the best TTL is for individual super-peers to monitor the response messages it receives. If a super-peer rarely or never receives responses from beyond x hops away, then it can decrease its TTL to x without affecting its reach.

When the desired reach covers just a subset of all nodes, however, finding a TTL to produce the desired reach is more difficult. Such a decision should be made globally, and not locally, since otherwise super-peers have an incentive to set the TTL of their queries unnecessarily high: to get more results at the expense of other nodes' loads.

The correct global TTL can be obtained by predicting the EPL for the desired reach and average outdegree, and then rounding up. Looking again at Figure 5.7, we see that when average outdegree is 20 and the desired reach is 500 nodes, the EPL is roughly 2.5. Hence, TTL should be set to 3. We must also note that setting TTL too close to the EPL will cause the actual reach to be lower than the desired value, since some path lengths will be greater than the expected path length. For example, when average outdegree is 10 and reach is 500, EPL is 3.0. However, setting TTL to 3 results in a reach of roughly 400 nodes, rather than 500.

In the absence of experimental data, a close approximation for EPL in a topology with an average outdegree d is $\log_d(reach)$. Such an approximation is most accurate for a tree topology where the query source is the root. In a graph topology, the approximation becomes a lower bound, as the "effective outdegree" is lower than the actual outdegree due to cycles. This approximation is useful for finding a good TTL without running experiments like the ones used to generate Figure 5.7.

Varying Configurations. For the query and join rates observed for Gnutella, the ratio of queries to joins is roughly 10 – that is, a user will submit an average of 10 queries during each session. This ratio is important because it determines the *relative load* incurred by each type of action. Relative load of queries and joins is important because it affects the relative performance of different configurations, depending on how "join-intensive" these configurations are. For example, the cost of joins is doubled with super-peer redundancy, while the cost of queries is halved. If joins are expensive relative to queries, this tradeoff may no longer be good.

Let us consider new configurations where the relative load of queries to joins is low. In particular, we will change the query rate listed in Table 5.1 to be $9.26 \cdot 10^{-3}$, such that the ratio of queries to joins is now roughly 1. We could have also decreased the relative load of queries by making the cost of queries less expensive; for example, modeling a system in which all queries are for rare items and therefore receive few results. The effects of both cases are similar.

Figure 5.9 shows the aggregate bandwidth of the same four systems shown in Figure 5.2, but with the lower query rate. We observe two effects: first, the aggregate load still decreases as cluster size decreases, but to a lesser degree than what we observed in Figure 5.2. Load decreases less rapidly because the savings in query communication overhead allowed by larger clusters is now small in comparison to the load of joins. Second, we observe that super-peer redundancy causes aggregate load to increase by a greater amount than before. For example, when cluster size is 100 in the strongly connected system, aggregate bandwidth increases by 14% with redundancy.

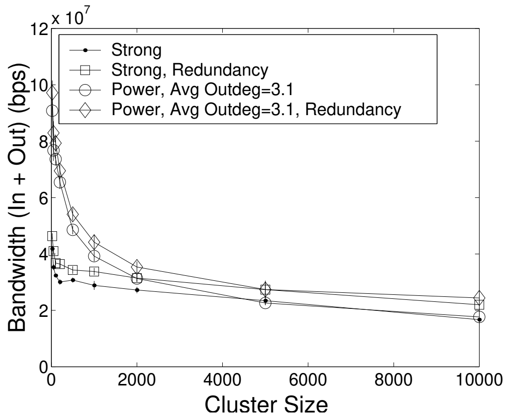

Figure 5.9: Aggregate load with lower query rate

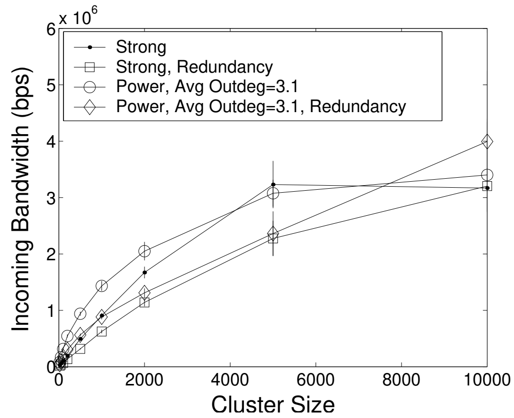

Figure 5.10: Individual load of super-peers with lower query rate

Figure 5.10 shows the individual incoming bandwidth of super-peers for the same four systems shown in Figure 5.3 (Section 5.3.1), but with the lower query rate. First, we observe that individual load now reaches a maximum at $ClusterSize = GraphSize$, since load is now dominated by joins. Second, note that super-peer redundancy does not decrease load as effectively as before (though the improvements are still good). When cluster size is 100 in the strongly connected network, individual incoming bandwidth load decreases by roughly 30% with redundancy, while processing load decreases by 18%. Recall that super-peer redundancy trades decreased query load for increased join load, which is not as helpful when the relative rate of joins is high. However, bear in mind that super-peer redundancy is still equally effective in improving reliability.

In summary, though the benefits of various cluster sizes and super-peer redundancy still apply, they may vary in degree based on the relative cost and rates of queries to joins.

5.3.2 Procedure for Global Design

Putting all these rules together, we now present a general procedure for the global design of a P2P topology, shown in Figure 5.11. The global design procedure is intended for a system administrator or designer, and its goal is to suggest an efficient system configuration given constraints and properties specified by the designer. These constraints include the maximum load and open connections allowed on an individual super-peer, and the maximum aggregate load on the system, if desired. The properties include the number of users in the network and the desired reach. (As mentioned before, the desired reach can be chosen according to

(1)	Select the desired reach r.
(2)	Set TTL=1
(3)	Decrease cluster size until desired individual load is attained. – if bandwidth load cannot be attained, decrease r, as no configuration can be more bandwidth-efficient than TTL=1. – if individual load is too high, apply super-peer redundancy, and/or decrease r.
(4)	If average outdegree is too high, increment TTL by 1, and go back to step (3)
(5)	Decrease average outdegree if doing so does not affect EPL, and reach r can still be attained.

Figure 5.11: Global Design Procedure

the desired number of results per query, since the two are proportional.) Although we can not prove the procedure yields the "best" topology, empirical evidence from analysis shows it usually returns a topology for which improvements can not be made without violating the given constraints.

The design procedure may be applied during *initial* design time, and it can be applied *incrementally* as the system is running. For initial design, the desired properties and constraints of the system, based on user behavior and peer characteristics, must be known at design time. Such information may be difficult to obtain accurately before the system runs. Hence, in addition to the initial design phase, a centralized decision-maker can run with the system and periodically re-evaluate the design procedure based on observation and feedback from the super-peers. For example, users joining the Morpheus P2P network must first contact a centralized server which then directs them to a super-peer. Though little is known about what this server does in the proprietary Morpheus system, it is feasible that this server, which is clearly in contact with the super-peers and has access to global information, can make decisions on the best topology based on the current workload, and send directives to the super-peers to achieve this topology. The next section discusses local decision making in the absence of design-time information and a centralized decision-maker.

Let us illustrate how the procedure works by using it to refine the topology used by today's Gnutella system. As mentioned in Section 5.2, analysis of network crawls show that the average outdegree for Gnutella is 3.1. At the time of this writing, the current Gnutella network size has been reported to vary between 10000 to 30000 users [56], so we choose 20000 to be size of the network in our analysis. Cluster size is 1, since there are few or no

	Incoming Bandwidth (bps)	Outgoing Bandwidth (bps)	Processing Power (Hz)	Number Results	EPL
Today	$9.08 \cdot 10^8$	$9.09 \cdot 10^8$	$6.88 \cdot 10^{10}$	269	6.5
New	$1.50 \cdot 10^8$	$1.90 \cdot 10^8$	$0.917 \cdot 10^{10}$	270	1.9
New w/ Red.	$1.56 \cdot 10^8$	$1.85 \cdot 10^8$	$1.01 \cdot 10^{10}$	269	1.9

Table 5.3: Aggregate load comparison between today's system, and the new design

super-peers, and analysis shows that the reach of such a topology with TTL=7 is roughly 3000 nodes out of 20000 total.

First, let us specify the desired constraints and properties of our system. To compare the results of the procedure with the current Gnutella system, we set reach to be 3000 peers (equivalently, $\frac{3000}{\text{cluster size}}$ super-peers), and the size of the network to be 20000 peers. For individual loads, let us limit our expected upstream and downstream bandwidth load to 100 Kbps each way, our expected processing load to 10 MHz, and our expected number of open connections to 100. Let us also assume for now that to keep the peer program simple, we do not want super-peer redundancy. Note that in specifying load limits, it is important to choose a limit that is far below the actual capabilities of the peer, for several reasons. First, actual load may exceed expected load during bursts of activity. Second, the expected load is for *search* only, and not for download, chat and other activities in the system. Finally, users will probably not want all their resources to be devoted to the system, but just a small fraction.

Now, let us run the procedure over our specifications. Reach has already been set to 3000 (step 1). Next, we set TTL=1, so that EPL=1 for most efficient bandwidth usage (step 2). Now we must choose a small enough cluster size such that our requirements for individual load are met (step 3). Under our constraints, analysis shows the largest supportable cluster size is roughly 20. However, to achieve the desired reach, average outdegree must be 150 (in order to reach $\frac{3000}{20} = 150$ super-peers), leading to a total of 169 open connections (an additional 19 connections for clients) – far exceeding our limit. Following the procedure (step 4), we increase TTL to 2, and select a new cluster size of 10 using analysis, which meets all our load requirements. To achieve our desired reach, each super-peer must have about 18 neighbors (since expected reach will be bounded above by roughly $18^2 + 18 = 342$). Looking at Figure 5.7, we see that decreasing outdegree affects our reach, so we will not change the outdegree (step 5). Total average outdegree is therefore $9 + 19 = 28$.

How does this new topology, generated by our procedure, compare with the old one? Table 5.3 lists the aggregate loads of the two topologies, along with number of results. The new topology achieves over 79% improvement over the old in all expected load aspects, while maintaining slightly higher expected quality of results. Furthermore, though we do not capture response time in our model, the average response time in the new topology is probably much better than in the old, because EPL is much shorter. Table 5.3 also lists the aggregate cost of the new topology with super-peer redundancy. As we can see, aggregate cost is barely affected by the introduction of redundancy.

Individual load likewise shows dramatic improvement. Figure 5.12 shows the outgoing bandwidth load of every node (super-peers and clients) in the system, for all three topologies. These values were taken from a single representative instance of each topology configuration. Comparing the new topology with and without redundancy, we see that the average super-peer load (that is, the top 20% ranked nodes in the topology with redundancy, and the top 10% ranked nodes in the topology without) is 41% lower with redundancy than without. As a tradeoff, average client load is 2 to 3 times greater with redundancy. However, since client loads are already very low – on the order of 100 bps – this load increase should be barely perceptible to the clients.

Comparing the new topology without redundancy to today's topology, Figure 5.12 shows that for the lowest 90% of loads in both system, individual load is one to two orders of magnitude lower in the new topology than in the old. This 90% corresponds to the 90% of nodes that are clients in the new topology. Among the highest 10% of loads, we still see significant improvement in the new topology over the old, ranging from 40% improvement at the "neck" occurring at about the 90th percentile, to a whole order of magnitude for the top .1% heaviest loads. Individual users will therefore have a much better user experience with these reduced loads – including those who agree to become super-peers, especially if super-peer redundancy is used.

5.3.3 Local Decisions

In the case where constraints and properties of the system can not be accurately specified at design time, and in the case where a centralized decision maker is not desirable or simply not available at the moment, super-peers should be able to make local decisions that will tend towards a globally efficient topology. Here, we will qualitatively discuss rules for local decision-making, based on the global rules from Section 5.3.1.

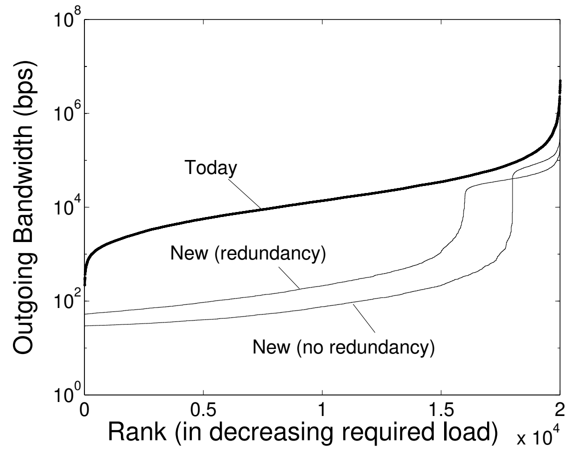

Figure 5.12: New design has much lower loads

In the following discussion, we assume that super-peers set a limit to the load they are willing to handle, and that they are equally willing to accept any load that does not exceed this limit (i.e., they will never refuse to increase their load unless it exceeds their predefined limit). The guidelines we will discuss are actions that will help *other* super-peers at the expense of the super-peer who follows them; hence, our "limited altruism" assumption allows us to suppose that every super-peer is willing to incur these expenses on themselves, which will ultimately result in less load for everyone. (Ongoing research in other contexts (e.g. [14, 36]) is exploring how to detect whether every peer in a P2P system is correctly playing their part, and punish the freeloaders and malicious nodes.)

I. A super-peer should always accept new clients. Given that the client must be served by some super-peer in the network, it generally does not make sense for a super-peer to refuse the client. If a super-peer finds that its cluster size is getting too large to handle (i.e., load frequently exceeds the limit), it should select a capable client from its cluster to become a partner as a redundant super-peer. Alternatively, it can select a client from its cluster to become a new super-peer, and the cluster should be split into two. In this manner, the number of super-peers can adapt for an increasing global workload. Likewise, if a cluster becomes too small (i.e., load remains far below the limit), the super-peer may try to find another small cluster, and coalesce the clusters. There is no set rule as to how small a cluster should be before the super-peer tries to coalesce, as there are always other uses for the extra bandwidth (e.g., accepting a new neighbor). Following these guidelines, according to global rule #1, clusters will tend to be as large as possible while respecting individual limits, thereby minimizing aggregate load.

II. A super-peer should increase its outdegree, as long as its cluster is not growing and it has enough resources to spare. By rule #3, increasing outdegree will decrease EPL, which is beneficial to the network as a whole. Increasing outdegree when TTL is small will increase the quality of results for the super-peer and its clients, at the cost of a proportionally higher load. However, if TTL is high (in particular, the TTL of the query messages forwarded by this super-peer), the super-peer should be cautious about accepting a new neighbor, because it may find itself forwarding a large number of messages for other nodes.

As we saw earlier, increasing outdegree is only effective if everyone participates. If a super-peer finds that it does not have enough resources to support more than a few neighbors, it should consider dropping some clients to free up resources, or “resign” to become a client itself.

III. A super-peer should decrease its TTL, as long as it does not affect its reach. For the case where the desired reach is not every super-peer in the system, a decrease in TTL should only happen after an increase in outdegree, in order to maintain the same reach. For the case where a node wants to reach all super-peers, according to rule #4, decreasing TTL may not affect reach, and should therefore always be done. A super-peer can tell if its reach has been affected by whether or not it receives fewer results.

5.4 Conclusion

Super-peer networks are effective because they combine the efficiency of the centralized client-server model with the autonomy, load balancing and robustness of distributed search. They also take advantage of heterogeneity of capabilities across peers. Because super-peer networks promise large performance improvements for P2P systems, it is important to understand their behavior and how they can best be used. In this chapter, we addressed this need by considering redundancy in super-peer design and topology variations, and carefully analyzing super-peer network performance. From our results, we have been able to extract rules of thumb that capture the essential tradeoffs, a global design procedure, and local decision-making recommendations for a globally efficient system.

Chapter 6

Non-Forwarding Search

Most of the research in unstructured P2P networks, including the previous two chapters, have focused on *forwarding-based* techniques, where a query message is forwarded between peers in the overlay until some stopping criterion is met. As we discussed in Chapter 1, different refinements of forwarding-based techniques have been studied, such as arranging good topologies for the overlay [16, 18], intelligent forwarding of messages within the overlay [17, 47, 91, 95], the use of lightweight indices, data replication [12], and many combinations of the above [10].

Despite the success of the above research in showing how forwarding-based techniques can be effective, some of the results also raise the question of whether message forwarding is truly necessary. For example, message-forwarding makes it difficult to control how many peers receive the query message, and which peers receive it, since there is no centralized point of control to monitor and guide the messages. However, we showed in Chapter 4 how incremental forwarding of query messages and intelligent peer selection greatly improves search performance without affecting quality of results.

In this chapter, we investigate a new type of search architecture, in which messages are *not forwarded*, and peers have complete control over who receives their queries and when. We study this non-forwarding architecture in the context of the GUESS [40] protocol, an under-construction specification that is meant to become the successor of the widely-used but inefficient Gnutella protocol. Under the GUESS protocol, peers directly probe each other with their own query messages, rather than relying on other peers to forward the message.

However, the GUESS protocol is being designed without a good understanding of the issues and necessary strategies to make it work. For example, when processing a query, in what order should peers be probed? The solution to this "peer selection" problem must balance efficiency of the query with load-balancing among the peers. Also, if messages are not forwarded, then a peer must know of many other peers (rather than just a handful of neighbors) in order to successfully find answers to its queries. How should this large state be built up and maintained? Practical problems not directly related to search performance must also be addressed; for example, since peers no longer rely on other peers to forward their queries, it is much easier for peers to abuse the system for personal gain. How can we detect and prevent selfish behavior? In this chapter we investigate solutions to these and other issues to make GUESS a viable alternative to other proven P2P search protocols.

We note that the non-forwarding concept has also been proposed for one-hop lookup queries in DHTs [42]. Like [42], the purpose of GUESS is to reduce the overhead of message forwarding; however, because GUESS allows "flexible" search over loosely structured networks, the protocol and its underlying issues (e.g., how to maintain state, how to select peers to query, etc.) are very different.

In this chapter, our goals are to examine the concept of a non-forwarding search mechanism for flexible search, understand what the tradeoffs are compared to existing forwarding-based techniques, and investigate how the GUESS non-forwarding protocol can be optimized. In particular, the contributions are as follows:

- We present an overview of the GUESS protocol (Section 6.1), based on the specification written by the Gnutella Development Forum [40].

- We identify the importance of *policies* in the performance of a non-forwarding protocol, and introduce several policies that are feasible to implement in a real system, and that might accomplish reasonable goals such as fairness, freshness and efficiency of search.

- Using simulations, we demonstrate how GUESS, if implemented in a straightforward way, can have serious performance problems. For instance, we show how careful choice of policy can improve performance dramatically (Section 6.5.2), but that a naive choice can result in a mechanism that is unfair (Section 6.5.3) and not robust (Section 6.5.4).

Most of the results presented in this chapter first appeared in [98].

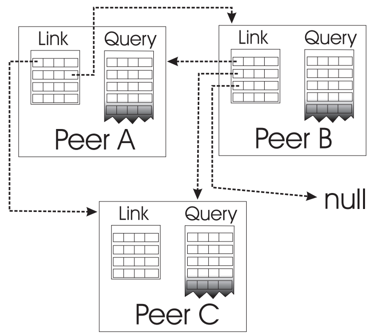

Figure 6.1: Illustration of a small GUESS network. Note that peer A points to peer C, but C does not point back to A; peer B has one entry pointing to a non-existing peer. Although neighbor pointers do not actually represent open, active connections between peers, they still form a "conceptual" overlay network, as illustrated in Figure 6.2

6.1 GUESS Protocol

In this section we describe the GUESS protocol for querying and state maintenance. For more details, please refer to the original specification [40]. Some of the information in this section is not part of the original protocol (e.g., the format of a cache entry), but are implementation details added for clarity.

6.1.1 Basic Architecture

Peers running the GUESS protocol will maintain two *caches*, or lists of pointers (IP addresses) to other peers: a *link cache* and a *query cache*. The link cache is analogous to a peer's neighbor list in Gnutella; all peers appearing in the link cache of a peer P can be considered P's neighbors. Rather than keeping an open TCP connection with each neighbor, however, P will communicate with neighbors via UDP. Hence, the "neighbor" relationship is one way: if Q appears in P's link cache, P might not appear in Q's link cache. Furthermore, because the UDP protocol does not maintain an active connection between two hosts, it is possible for a peer's neighbor to die without the peer's knowledge. We discuss the issue of maintaining neighbor pointers in Section 6.1.2. Please refer to Figure 6.1 for an illustration of a GUESS network.

The *query cache* is simply a "scratch space" to temporarily hold large numbers of pointers to other peers in order to improve query performance. We discuss the use of the query cache further in Section 6.1.3.

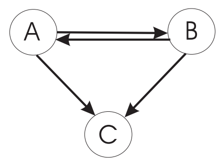

Figure 6.2: Conceptual overlay representation of the GUESS network in Figure 6.1

An *entry* in the link or query cache, essentially a "pointer" to some peer Q, has the following format:

$$\{\text{IP address of } Q, TS, \mathit{NumFiles}, \mathit{NumRes}\} \tag{6.1}$$

The TS field holds the timestamp of the last interaction with peer Q. When P interacts with Q, regardless of which party initiated the interaction, P will update the TS field in its cache entry for Q, if such an entry exists. The *NumFiles* field holds the number of files being shared by Q. This field is set by Q when it first "introduces" itself to the network, and is passed on as cache entries are shared (introduction and cache entry sharing are discussed in Section 6.1.2). Similarly, the *NumRes* field holds the number of results last returned by Q. Each time peer P sends a query to peer Q, it resets the value of NumRes according to Q's response to that query.

6.1.2 Maintaining State

Because peers in P2P systems typically have short lifetimes [82], peers must actively make sure the entries in their link cache are fresh. Otherwise, over time a peer's link cache will accumulate the addresses of many dead peers, which can result in poor query performance for that peer, and fragmentation of the "conceptual overlay."

A peer maintains its link cache by periodically selecting an entry and sending a *Ping* message to the neighbor. If the neighbor does not respond, the peer will evict this entry from its cache. If the neighbor does respond, then the peer will update the TS field of the cache entry. Note that given a fixed effort from the peer to maintain its link cache, the rate at which a given cache entry is pinged is inversely proportional to the size of the cache. Therefore it is important that the cache not be too large; otherwise, it cannot be properly maintained.

When a peer receives a Ping message, it will respond with a *Pong* message. A Pong message contains a small number of IP addresses selected from the peer's own link cache. The purpose of Pong messages is to allow peers to *share* cache entries with each other.

Sharing entries helps peers discover new live, productive peers to place in their cache. When a peer receives a Pong message, it will decide whether to add some or all of the entries to its link cache, depending on the cache replacement policy in use. If the peer does decide to place an entry from the Pong message into its own cache, it does not update any of the fields (i.e., TS, NumFiles, or NumRes).

When a new peer first joins the network, it does not appear in any other peers' link caches. An *introduction protocol* is needed to bring the IP address of new peers into the link caches of existing peers. For our purposes, we assume that when a peer P initiates an interaction with peer Q by sending either a Ping or Query message, then Q will add P to its cache with some probability p. Note that it is important that $p < 1$; otherwise it would be very easy for malicious peers to infiltrate the caches of many good peers (see Section 6.5.4 for a discussion of "cache poisoning"). A new peer will therefore be added to existing peers' caches with some probability as soon as it initiates a query or ping. Once the new peer appears in other peers' caches, it can be circulated even further via Pong messages.

6.1.3 Query Propagation

The essential characteristic of GUESS is that Query messages are not propagated via flooding-based broadcast. Instead, a GUESS peer simply iterates through the entries in its link cache, and performs a unicast query, or *probe*, to the target peer. A peer should probe only as many other peers as necessary to obtain a sufficient number of results. Also, the GUESS protocol specifies that querying happens in a strictly serial manner; after sending a probe, a peer must either receive the reply or wait for a *timeout period*, before it may probe the next neighbor.[1]

In some cases, a querying peer must be able to probe a large number of peers to receive satisfactory results. However, the number of addresses that a peer can actively keep track of is limited by the size of the link cache. As we discussed in the previous section, the link cache must be relatively small if we are to maintain it properly.

To counter this problem, when a peer is probed, it returns a Pong message in addition to any results to the query it may find. Then, the querying peer places the entries from this Pong message in its *query cache*, which is a temporary cache of (theoretically) unbounded size. Entries in the query cache have the same format as link cache entries. The querying peer may probe peers from either its link cache or its query cache. In this manner, a peer is

[1]Parallel probes are also possible, although the current GUESS specification makes no such provisions.

able to probe a much larger number of peers than it can maintain in its link cache. Entries in the query cache are not maintained after the query is completed, otherwise, maintenance overhead would be too high. However, qualifying entries may be inserted into the link cache, depending on the cache replacement policy in use.

6.2 GUESS vs. Gnutella

To understand the advantages and drawbacks of the non-forwarding GUESS protocol at a high level, we would like to compare it to Gnutella, the predecessor to GUESS. Gnutella is a forwarding-based protocol, and is also the closest existing protocol to GUESS in terms of functionality and intended use. Unfortunately, a direct comparison between GUESS and Gnutella is difficult because a great deal of research has been done to improve Gnutella (e.g., see Section 4.1). GUESS on the other hand is relatively new and there are many possible improvements that might be made. Furthermore, it is not feasible to implement all of the possible optimizations for each protocol to produce a quantitative comparison, nor is it reasonable to compare implementations of the basic versions of the protocols which have some obvious shortcomings. Hence, in this section we present a high-level, *qualitative* comparison of the efficiency and security characteristics of the two protocols. We attempt to focus on the fundamental characteristics of GUESS and Gnutella, which will hold true despite optimizations on either protocol. In Section 6.5, we perform an in-depth quantitative analysis of the GUESS protocol only.

6.2.1 Query Performance

Perhaps the primary point of comparison between GUESS and Gnutella is that of efficiency of queries, in terms of network resources used. In Gnutella, the location and extent of a query are determined by the overlay topology; queries will be received by whichever peers happen to be within a certain radius of the originator. Because the execution of a query is not adaptive, it is often inefficient. For queries for popular items, Gnutella returns far too many results; for queries for rare items, Gnutella does not return enough results. Furthermore, although different peers may have different needs (e.g., search for different types of content), the basic Gnutella protocol does not differentiate between peers.

In contrast, the GUESS search protocol provides each client with fine-grained control over the execution of the query. In particular, peers can control: (1) the **order** in which other peers are probed, and (2) the **extent** of a query. By controlling the *order* of probes, a peer in GUESS can use metadata or past experience to identify other peers that can quickly return results. In Section 6.5.2 we will see how the order in which peers are probed has an enormous impact on query performance. One drawback to ordering probes is the possibility of hotspots, where many peers all try to probe the same productive peer, and that productive peer is overloaded. This issue is further explored in Section 6.5.3. We also note that to a limited extent, techniques such as routing indices [17] and directed breadth-first-search (Chapter 4) allow peers in Gnutella to control the order in which peers receive their query.

By probing peers iteratively (rather than all in parallel), a peer can adapt the *extent* of a query to the popularity of the item being sought, thereby resulting in fewer wasted probes, and better satisfaction results. If a file is popular and adequate results are obtained quickly, a search can be terminated after querying a small number of peers. Alternatively, if a file is rare, the search can continue over a much larger number of peers until the search is satisfied. Again, to a limited degree, techniques such as iterative deepening (Chapter 4) allow peers to control the extent of the query. However, the degree of control is much more coarse-grained than in GUESS, thereby resulting in greater inefficiency.

One important drawback to probing peers iteratively is poor *response time*. For example, if 1000 peers are probed before the query is satisfied, and peers must wait .2 seconds between successive probes, the query will be satisfied only after 200 seconds – possibly too long a time to satisfy the user. In contrast, because all peers in Gnutella are probed in parallel (more or less), response time for Gnutella is very good. Response time in GUESS may be improved by parallel walks (probing a small number of peers in parallel), or adapting the rate of probes according to how many peers have already been probed. For example, for every 10 seconds that pass by without finding a result, the rate of probes may be doubled. Furthermore, if the order in which peers are probed is carefully chosen, response time may actually be better in GUESS than Gnutella, in some cases. Again, response time in GUESS is further explored in Section 6.5.

6.2.2 State Maintenance

Although GUESS is generally superior to Gnutella in terms of query efficiency, it is inferior to Gnutella in regards to the cost and complexity of state maintenance. A Gnutella peer maintains a small number of open network connections to its neighbors in the overlay network. In contrast, a GUESS peer must maintain pointers in a fairly large pong cache. Although the size of state is generally not an issue given the availability and low cost of memory, the amount of network traffic needed to maintain the cache can be high if the cache is large. In Section 6.5.1 we investigate the cost of maintaining state in greater detail.

Related to the above concept of state is the consistency of the network over time. In Gnutella, because state is localized to the immediate neighbors of each peer, the process of joining and leaving the network is simple. A peer notifies its neighbors when it joins or leaves the network, and only these neighbors need to modify their state to maintain a consistent view of the network. In contrast, a peer in GUESS may belong in the link caches of many other peers, but because neighbor relationships are 1-way, it does not have knowledge of who these other peers are. Hence, when a peer leaves the network, it leaves without notifying anyone. Other peers must then discover the death of that peer with an unsuccessful probe, which results in wasted work. Furthermore, when a peer joins the network, it must actively introduce itself into other peers' pong caches, before it can be of any use to the network.

6.2.3 Security

Because peers in a large-scale P2P system are autonomous and may come from competing organizations, a good search protocol must be robust against misbehaving peers. We define two types of misbehavior: *selfish* behavior, and *malicious* behavior.

Selfish Peers. A selfish peer tries to "game" the system in order to maximize its own utility, regardless of the cost it imposes on other peers. It does not desire to bring down the system, since that would cause poor utility to the selfish peer as well. With a search protocol like GUESS or Gnutella, a selfish peer will attempt to maximize the number of results it receives, while minimizing response time.

In Gnutella, it is difficult for a single peer to game the system, because peers do not have control over the execution of their own query. If a peer wishes to receive more results for its queries in less time, then it must connect to more neighbors. Having more neighbors increases the load on that peer, due to the forwarding traffic in Gnutella; hence, in order to improve its search performance, a peer must first contribute greater resources to the system. Although a peer may drop messages in order to reduce its load, it is a simple matter for its neighbors to observe that they rarely receive traffic (e.g., query response messages) from the selfish peer, and to disconnect from that peer in return.

In GUESS, it is very easy for a peer to game the system, because peers have complete control over the execution of their own queries. Rather than iteratively probe peers on a query, a selfish peer can simply probe thousands of peers at a time. This behavior may incur a much higher load on other peers (e.g., if the query could have been answered with 20 probes), while possibly drastically improving the response time of the search for the querying peer. If all peers act according to their best interests, the system might fail as if under a denial of service (DoS) attack. Therefore, in order for GUESS to work, there must be a way to give peers an incentive to adhere to the protocol. One straightforward proposal is to have peers "pay" for each probe. Peers will then be motivated to probe as few peers as possible to answer their queries. Such a solution does require a payment mechanism, such as [97]. Other ways to limit selfish behavior remains an open problem.

Malicious Peers. The goal of a malicious peer is to make the system unusable. For the purposes of comparison, we will focus on attacks on the protocol itself; we ignore attacks that are unrelated to the choice of protocol, such as returning inauthentic documents. We will discuss malicious behavior further in Section 6.5.4, where we will also consider ways to improve the robustness of the GUESS protocol.

The Gnutella protocol is susceptible to application-level *denial-of-service* (DoS) attacks, due to the "amplification effect" inherent to the protocol. Because queries are flooded across the network, a malicious peer needs to expend little effort to cause a much higher load on the system. Reference [21] presents some techniques for preventing DoS attacks in Gnutella, though naturally, these preventative measures result in lower efficiency in the case of cooperative behavior.

Some studies (e.g., [82]) also suggest that Gnutella is susceptible to *fragmentation* attacks, where highly-connected peers are brought down via network-level DoS attacks, and the overlay topology is fragmented. Fragmentation is bad because it can result in worse quality of response (e.g., fewer responses) for queries. However, this weakness is not inherent to the Gnutella protocol itself, but is an artifact of the type of topology (power-law) that naturally arises from peers' local connection decisions [90]. The Gnutella network can be made more robust, for example, by imposing simple limits on the number of connections peers are allowed to make.

The GUESS protocol is not as susceptible to application-level DoS attacks, because query messages are not amplified – the malicious peer will need to expend effort proportional to the number of messages sent. However, as discussed earlier, many peers acting selfishly can induce a similar effect.

On the other hand, the GUESS protocol is susceptible to fragmentation attacks, if malicious peers collude. It is fairly easy for a malicious peer to insert itself or other malicious peers into good peers' link caches, via Pong messages. If a large percentage of many peers' link caches are filled with malicious peers, and all malicious peers disappear from the system at the same time, then the conceptual overlay formed by peers' link caches will become fragmented. Such behavior by malicious peers is known as "cache poisoning." The effects of cache poisoning and prevention are discussed further in Section 6.5.4.

Finally, we note that other security-related issues, such as privacy and anonymity, are easier to achieve in Gnutella than in GUESS. Because peers control their own queries in GUESS, there is no way for a peer to hide its query behavior from other observing peers. Therefore, GUESS may not be appropriate for certain privacy-sensitive applications.

In summary, giving peers control over the execution of their query, while potentially improving efficiency of the system, also opens up the possibility of abuse and manipulation. Before an actual deployment of GUESS can give such control to peers, we must (1) learn how peers can best use this control to optimize query performance, (2) add other mechanisms or incentives to enforce correct behavior, and (3) learn how peers can avoid being manipulated by malicious peers.

6.3 Policies

We observe that performance of the GUESS protocol depends heavily on the *policies* that determine how entries in the pong cache are used and maintained. For example, by constructing a Pong message using entries with the latest timestamps, and evicting entries with the oldest timestamps, peers might be able to maximize the number of live entries in their cache. As another example, by first probing peers who have a history of providing useful information, peers can drastically reduce the total number of probes needed to answer a query. Hence, any deployment of the GUESS protocol must first carefully consider which policies to implement.

There are five *types* of policies that we must consider:

- `QueryProbe` – the order in which peers in the link and query caches are probed for queries

- `QueryPong` – the preference given to entries when constructing a Pong message in response to a Query
- `PingProbe` – the order in which peers in the link cache are pinged
- `PingPong` – the preference given to entries when constructing a Pong message in response to a Ping
- `CacheReplacement` – the order in which peers are evicted from the link cache

We consider QueryProbe and PingProbe (and QueryPong and PingPong) separately because a peer might have different goals during a query and during a ping. For example, during a query, a peer may prefer to probe other peers who are likely to have a file. For a ping, however, a peer may prefer to probe other peers who have many link cache entries, or are known to be alive.

For each of the policy types discussed above, many policies could be implemented. For the purposes of our experimentation, we came up with a number of policies that we felt would be feasible to implement in a real system, and that might accomplish reasonable goals such as fairness, freshness and efficiency of search. The policies that we implemented are listed below along with a brief discussion of the rationale for each policy.

Note that for Cache Replacement, the policy name indicates which peers are evicted from the cache. Therefore, to achieve the same intended effect as, say, a Probe policy, we must reverse the criterion used. For example, to effect a Most Files Shared goal, we use a Cache Replacement policy of Least Files Shared, since by evicting those peers with a small number of files we retain the ones with more files. Similarly, Most Results becomes Least Results, and Least/Most Recently Used become Most/Least Recently Used.

Random (Ran) – selects entries at random. This policy is used as a baseline for comparison, and is likely to be very fair in terms of load distribution.

Most Recently Used (MRU) – prioritizes link cache entries with the most recent timestamps. These entries are most likely to be alive since they have been in contact recently; therefore MRU should waste the least amount of work in probing dead peers.

Least Recently Used (LRU) – opposite of MRU, prioritizing cache entries that have old timestamps. The rationale behind LRU is *fairness*; rather than continually querying the same set of peers, load is spread across peers that have not been contacted recently.

Of course, peers with very old timestamps are more likely to be dead, resulting in wasted probes.

Most Files Shared (MFS) – prioritizes entries based on the number of files they share. The impetus for such a policy is obvious; peers with many files are more likely to be able to have files related to the query. A potential downside to this policy is that the measure used (files shared) is global, and so the peers with many files shared are likely to be queried by a large number of peers, making them shoulder an unfair amount of work in the network.

Most Results (MR) – similar in nature to Most Files Shared, prioritizes entries based on the number of good results that the corresponding peers have returned in the past. Typically, peers that have been fruitful in the past may be more likely to be good in the future. MR is less susceptible to lying than MFS, although often less good at identifying useful peers.

One potential advantage of MR over Most Files Shared is that MR includes some notion of *personal* usefulness. A peer with many files may not have the files that I want; however, if the queries that I generate are related, then perhaps a peer that has worked well for me will continue to work well, regardless of its total number of files. Similarly, MR might be better than MFS at identifying peers who are actually capable of servicing queries, which is important when capacity limits come into play.

6.4 Experimental Setup

We will be comparing different *configurations* of the GUESS protocol, where a configuration is defined by a set of system and protocol parameters, shown in Tables 6.1 and 6.2, respectively. System parameters describe the nature of the system on which the GUESS protocol is used (e.g., the query behavior of the users). Protocol parameters then describe how the GUESS protocol is configured (e.g., the policy used to order query probes). As we will see in Section 6.5, different protocol parameters result in better performance in different system scenarios. Parameters will be described in further detail as they are used later. Unless otherwise specified, our simulations use the default values shown in these tables.

Note that NetworkSize=1000 is a modest number of peers, given the scale of the types of system we expect to use GUESS. In our results section, we show how our results scale with network size, when appropriate.

Name	Default	Description
NetworkSize	1000	Number of peers in the network
NumDesiredResults	1	Number of desired results per query
LifespanMultiplier	1	Parameter used to vary peer lifespans
Query Rate	$9.26 \cdot 10^{-3}$	The expected number of queries per user per second
MaxProbesPerSecond	100	Maximum number of probes per second a peer is willing or able to handle
PercentBadPeers	0	The percentage of peers in the network that are malicious
BadPongBehavior	Dead	The type of IP address a bad peer returns in a pong (Dead, Bad, or Good)

Table 6.1: System parameters and default values

Metrics. The main metric of query efficiency is the average number of *probes per query* needed; minimizing probe traffic is one of the primary goals of the GUESS protocol. Of course, such a goal is only reasonable if users receive results for their queries, hence another key metric is the proportion of queries that go unsatisfied. We also examine the proportion of probes that are *wasted*; that is, sent to peers that have already left the network. For each peer, we also measure the number of *received* probes; comparing the load across peers helps us to gauge fairness as well as efficiency. Note that unlike in past chapters, here we measure probes, rather than the exact bandwidth and processing power consumed by sending and processing probes, because we wanted to clearly differentiate between good, dead and wasted probes. These concepts are unique to GUESS, and have therefore not been applied in previous chapters.

Simulation Framework. In a given simulation run, we simulate `NetworkSize` peers participating in the GUESS protocol. At time 0, all peers are alive. Each link cache is seeded with `CacheSeedSize` living peers. We found that as long as `CacheSeedSize` was small (e.g., approximately `NetworkSize`/100), the value of `CacheSeedSize` did not affect performance results.

As time progresses, peers will die. A large sample of peer lifetimes in the Gnutella network was measured in [82]. For our simulations, the lifetime of a peer is drawn randomly from this sample. In addition, we may tune these lifespans via the `LifespanMultplier` parameter. If `LifespanMultiplier` $= x$, then all values in the measured distribution of lifespans are multiplied by x.

When a peer dies, we assume that it never returns to the system. This assumption is conservative in that it is the worst-case scenario for cache maintenance; our maintenance policies must be shown to be effective even in this worst case. Also, when a peer dies, a new

Name	Default	Description
QueryProbe	Random	The policy used to determine which peer to probe for a query
QueryPong	Random	The policy used to determine which IP addresses to return in a pong
PingProbe	Random	The policy used to determine which peer to probe for a ping
PingPong	Random	The policy used to determine which IP addresses to return in a pong
CacheReplacement	Random	The policy used to determine which peer to evict from the link cache
PingInterval	30s	Elapsed time between pings
CacheSize	100	Size of the link cache
ResetNumResults	No	Flag indicating whether to reset the responses field in a cache entry
DoBackoff	No	Flag indicating whether to perform backoff
PongSize	5	Number of IP addresses per pong
IntroProb	.1	Probability of adding to your cache a peer who probes you.

Table 6.2: Protocol parameters and default values

peer is "born." In this way, there are always `NetworkSize` live peers in the system. When a peer joins the network, it must populate its link cache. We use the *random friend* seeding policy as described in [22]. Under the random friend policy, we assume that the new peer knows of one other peer, or "friend," that is currently alive. The new peer initializes its link cache by copying the link cache of its friend. A friend can be the IP address of a peer that the new peer saved from when it was last online, or peers can obtain the IP address of a friend from a "pong server." A pong server (e.g., LimeWire [56] for the Gnutella network) is a service that keeps track of live peers in the system. Because such a service is centralized and potentially expensive (e.g., if it receives many requests), we do not wish to make heavy use of the service. In particular, it is not reasonable for each new peer to ask the pong server to initialize its entire link cache with the IP addresses of many live peers.

The generation of queries at each peer follows a bursty pattern, in which a number of queries (number uniformly chosen between 1 and 5) are submitted in succession, followed by a long wait. The arrival of bursts follow a Poisson process, and the overall rate of queries per user is given by `Query Rate`. When a peer has a query, it will iteratively probe the peers in its link cache, which is of size `CacheSize`, until `NumDesiredResults` results are returned. The order in which the peer probes other peers is determined by the `QueryProbe` policy. For the purposes of our simulation we assume that the timeout period between probes is long enough for a live peer to respond within the period. In real life some peers may take longer than the timeout period; however, this would simply mean that a few unnecessary probes are performed, and would not significantly alter our results.

When a peer is probed, and it is not *overloaded* (described below), it will see if any files in its collection answer the query. To determine whether a peer returns a result for a query, we use the query model developed in Chapter 2. The probability of a peer returning a result depends partially on the number of files owned by that peer; number of files owned by peers are assigned according to the distribution of files measured by [82] over Gnutella. Whether or not a match is found for the query, the responding peer returns a Pong message with `PongSize` IP addresses to the querying peer. The IP addresses are selected from the link cache according to the `QueryPong` policy. When the querying peer receives the Pong, it will possibly insert the new IP addresses into its link cache, according to the `CacheReplacement` policy. The peer will also add the IP addresses to its query cache, if they have not already been seen before.

Peers set a maximum number of probes per second that it is able or willing to handle, according to `MaxProbesPerSecond`. Although different peers may have different capacities, we assume that in the interest of fairness, all peers set the same load limit. A peer is *overloaded* if the number of probes it must process per second is greater than `MaxProbesPerSecond`. When a peer becomes overloaded, it "refuses" a probe, meaning it notifies the querying peer that it is overloaded, and that the querying peer should "back off" from probing it. We will discuss backing off in further detail in Section 6.5.3.

Peers maintain the IP addresses in their link cache by pinging one peer every `PingInterval` seconds. The order in which peers are pinged is determined by the `PingProbe` policy. As with query probes, when a peer receives a ping, it will respond with a Pong message containing `PongSize` IP addresses. The IP addresses are selected from the link cache according to the `PingPong` policy. When the querying peer receives the Pong, it will possibly insert the new IP addresses into its link cache, according to the `CacheReplacement` policy.

The `IntroduceProb` parameter sets the probability of adding a peer to one's cache when that peer initiates an interaction, according to the introduction policy in use (discussion of introduction policies is found in Section 6.1.2). The parameters `PercentBadPeers` and `BadPongBehavior` describe malicious peer behavior, and are discussed further in Section 6.5.4.

6.5 Results

In the following section, we present the results of our experiments over a wide range of system and protocol configurations. We organize the results into four main categories. First, we

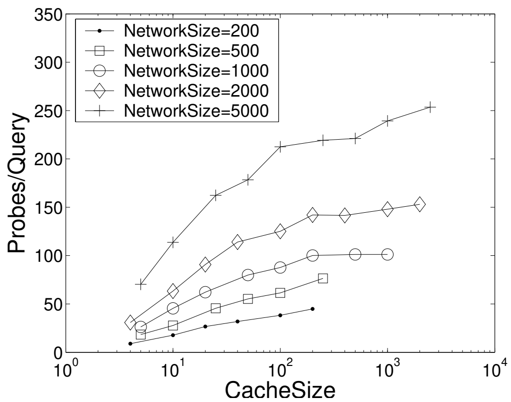

Figure 6.3: Number of probes increases as cache size increases

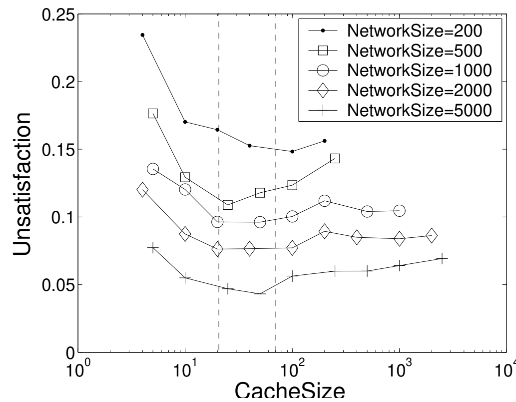

Figure 6.4: Unsatisfaction experiences a minimum at moderate cache values

investigate the issue of maintaining the link cache (Section 6.5.1) in the face of frequent peer downtimes. We then study the behavior of policies in various system scenarios. In Section 6.5.2, we study basic performance in the default usage scenario. We then investigate the fairness of policies and their ability to perform in the presence of limited peer capacity (Section 6.5.3), as well as the robustness of policies to misbehaving peers (Section 6.5.4).

6.5.1 Maintaining the Link Cache

Two of the most important issues in maintaining the link cache is the size of the cache, and the rate at which the cache is maintained via Pings.

Cache Size. Intuitively, very small cache sizes result in poor performance, because peers do not have enough "neighbors" to probe. Hence, queries will often not be satisfied. On the other hand, it is not immediately clear whether very large cache sizes will result in good performance, because of the effort necessary to maintain the freshness of the entries.

To investigate the impact of cache size on performance, Figures 6.3 and 6.4 look at query performance as cache size is varied, over a number of different network sizes ranging from 200 to 5000. Due to limitations in our simulator we do not scale beyond 5000. We study cache sizes ranging from 5 (very small) to the size of the network. To provide additional "strain" on the system, we set `LifespanMultiplier` to .2.

In Figure 6.3, we see immediately that as the size of the cache grows, regardless of network size, the number of probes per satisfied query also grows. A reasonable explanation for this phenomenon immediately comes to mind: as the size of the cache increases, there

Cache Size	Fraction Live	Absolute Live
10	.822	8.0
20	.759	14.8
50	.605	28.5
100	.418	36.2
200	.330	41.9
500	.309	41.9

Table 6.3: Breakdown of live cache entries for varying CacheSize values

are more peers to probe. Hence, many of the queries that were previously unsatisfied will now be satisfied, albeit at a higher number of probes. The consequence of this explanation is that as cache size increases, the unsatisfaction rate decreases.

Surprisingly, however, we see in Figure 6.4 that this explanation does not hold. As we expected, unsatisfaction is high when cache size is very small. However, across all network sizes shown, unsatisfaction experiences a minimum at a moderate cache size, after which point it increases once more. In other words, very large cache size not only results in more expensive queries, it also results in lower satisfaction.

To understand this phenomenon, Table 6.3 looks at the average fraction of cache entries that are live, as well as the average absolute number of live cache entries, for various cache sizes (using the default NetworkSize = 1000). Note that the total number of cache entries may be less than the cache size itself. The CacheSize parameter only tells us the capacity of the cache. Often caches are not full because entries are evicted if they are found to be dead.

Let us first consider the fraction of live cache entries (column 2) in Table 6.3. Here we see that the fraction of live entries greatly decreases as the size of the cache grows (although absolute number of live entries increases). The fraction of live entries is important because, given a Random probe policy, it roughly governs the fraction of probes to live peers (i.e., "good" probes), and the fraction of probes to dead peers (i.e., "dead" probes).

The effect of this ratio can be seen in Figure 6.5, where we take a closer look at the number of probes required to answer a query. For clarity, we look only at our default network size of 1000. The probes are broken down into "good" and "dead" probes. We see from Figure 6.5 that as cache size increases, the number of *dead* probes increases dramatically at first, and then levels off. The number of good probes, however, does *not* increase. Hence, although larger cache sizes result in a larger number of probes, they do not translate to more satisfied queries, because the additional probes are all useless.

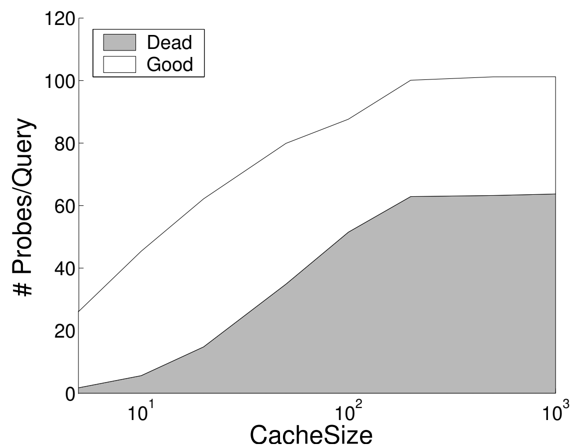

Figure 6.5: Number of dead probes increases as cache size increases, while good probes experience a maximum at moderate cache value

In fact, the number of good probes experiences a local maximum at a cache size of 20. At this maximum, the number of good probes is almost 30% higher than when cache size is 200. Since the satisfaction of a query depends only on the number of good probes, we can understand why satisfaction also reaches a maximum at a cache size of 20 (Figure 6.4). At larger cache sizes, the link cache maintenance effort is "spread too thin" over a large number of entries; hence, fewer entries are good when a query is executed. In addition, because Pong messages will contain more dead IPs if peers' link caches contain more dead IPs, the total number of useful *query cache* entries will also decrease proportionally.

In terms of how to select a cache size, clearly, moderate cache sizes have the best cost/performance tradeoff. In fact, in Figure 6.4, we see that a cache size in the range of 20-70 (range marked by dotted lines) results in best satisfaction – for all network sizes studied, the optimal cache size does not change with network size. Intuitively, optimal cache size might grow with network size, and experiments with very large network sizes (e.g., a million) should make this apparent. However, our experiments show strong evidence that optimal cache size grows very slowly, if at all. Hence, even with very large network sizes, a reasonably small cache size is sufficient for good performance.

Ping Interval. A fragmented overlay is bad because it degrades query performance, and unless there is some form of centralized boot-strapping server (e.g., pong servers such as those run by LimeWire [56] for Gnutella), the network is unlikely to heal. Here, we study how PingInterval must be set in order to maintain a *connected* topology. All cache sizes considered in our study thus far are large enough to maintain a connected topology, assuming cache entries are well-maintained. In order to keep a well-maintained cache, PingInterval may need to be adjusted as CacheSize and NetworkSize vary.

We note that the cache is maintained via Ping messages, and also via query messages, since peers respond to queries with both query responses and Pong messages. To isolate the effect of Pings, in the remainder of the section we do not simulate queries.

Figure 6.6 shows us the size of the largest connected component in a GUESS "conceptual overlay" for various cache sizes, as PingInterval varies across the x-axis. A fully connected network will have the largest connected component of size 1000. As expected, the smaller the PingInterval, the more connected the overlay is. As PingInterval increases, the overlay begins to fragment.

Comparing across the curves in this figure, we see that the smaller the cache size is, the smaller PingInterval must be. That is, smaller cache sizes must be more closely maintained. At first this result may seem surprising when compared to our results in the previous section, where smaller cache sizes result in better query performance. The reason for this difference is that for query performance, the *ratio* of live to dead peers is critical, whereas for connectivity, the *absolute number* of live peers is more important. Returning to Table 6.3, we see that although smaller cache sizes can have a dramatically higher fraction of live entries than large cache sizes, the absolute number of live cache entries is larger for large cache sizes.

In terms of how NetworkSize affects PingInterval, we found that for moderate cache sizes (e.g., between 20 and 100), the connectivity of the overlay is largely unaffected by network size. Figure 6.7 shows the relative connectivity of the GUESS overlay as PingInterval varies along the x-axis, for various network sizes. In this figure, CacheSize = 20, regardless of NetworkSize. We see from Figure 6.7 that the relative size of the largest connected component is roughly the same across network sizes, for a given PingInterval. Although one might expect that larger network sizes require more links per peer to remain connected, this is in fact not the case; each node in a random graph requires a small constant number of links for the graph to be connected with high probability.

In conclusion, there does not appear to be a set rule for good PingIntervals, but this parameter may be adjusted at runtime. After selecting a cache size that results in good query performance, a peer should adjust its PingInterval to maintain a certain threshold of live entries in its cache. While sending query or Ping messages, if a peer discovers that

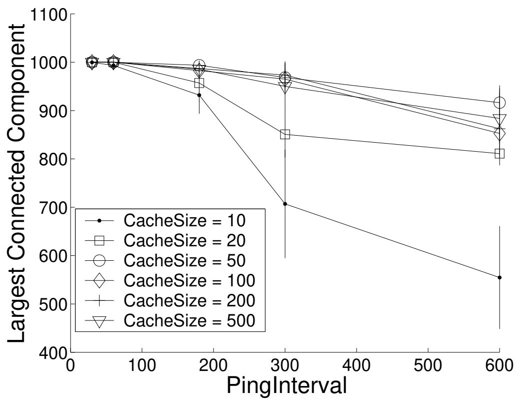

Figure 6.6: Small cache sizes are most negatively affected by long ping intervals

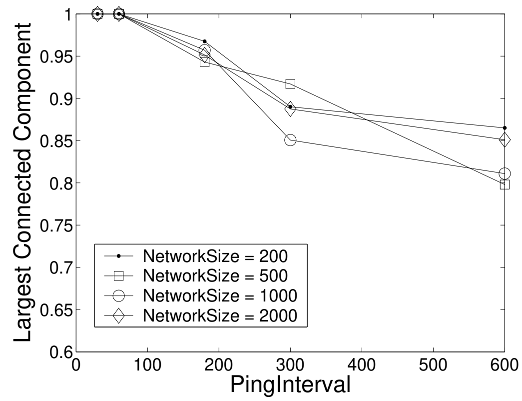

Figure 6.7: Selection of ping interval is largely independent of network size

many of its probes are to dead addresses, the peer should decrease its PingInterval. On the other hand, if a peer discovers that almost all its entries are live, then it may increase its PingInterval, if the current overhead of Ping/Pong messages is placing a heavy burden on its resources (which is possible, though unlikely). Given these guidelines, along with the guidelines for selecting a cache size, peers can maintain the health of the network with a very reasonable overhead.

6.5.2 Basic Policies

In this section, we study the basic performance of the GUESS protocol, including a brief comparison against Gnutella, and an in-depth comparison of policy performance.

Flexible Extent. First, we highlight the performance benefits of a completely flexible query extent. Figure 6.8 shows the tradeoff curve between the average cost of a query and the unsatisfaction rate for three different search mechanisms: a fixed-extent mechanism (e.g., Gnutella), a coarse-grained flexible extent mechanism (e.g., the iterative deepening approach in Chapter 4), and a fine-grained flexible extent mechanism (e.g., GUESS). With regards to extent, iterative deepening (shown by a square in Figure 6.8) is conceptually similar to GUESS, except that many peers (e.g., hundreds) are probed in each iteration, instead of just one. The 'o' and 'x' mark the points in the figure that represent GUESS, using the Random baseline policy and `QueryPong = MFS`, respectively. Finally, for the fixed-extent mechanism, we evaluated the rate of unsatisfied queries for all fixed extent sizes from

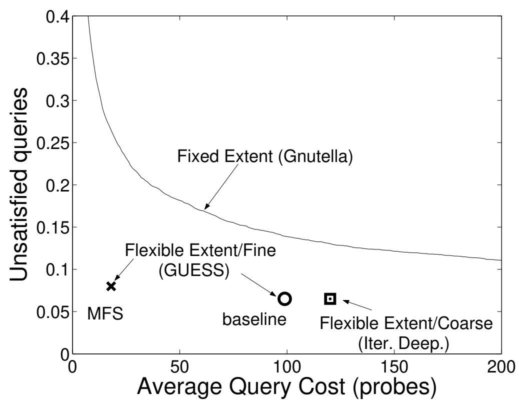

Figure 6.8: For a given average query cost, unsatisfaction rate is lowest with a fine-grained flexible extent provided by GUESS

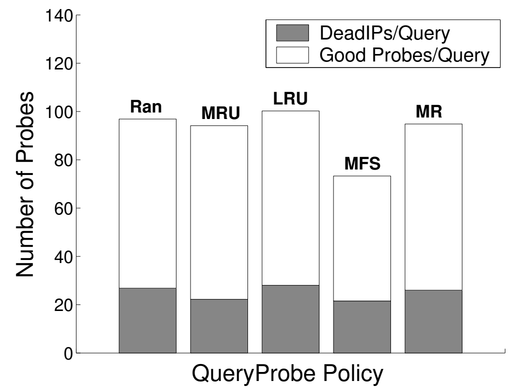

Figure 6.9: Probes/Query for different QueryProbe policies

1 to 1000, in order to view the tradeoff between efficiency and quality.

From Figure 6.8, we can see the enormous performance gains that a flexible extent allows. For example, GUESS can achieve an unsatisfaction rate of almost 6% with an average query cost of 99 probes using the baseline Random policy, and 8% unsatisfaction with an average of 17 probes with `QueryPong = MFS`. In contrast, a fixed extent mechanism such as Gnutella would require 1000 probes for 6% unsatisfaction, and 540 probes for 8% unsatisfaction – well over an order of magnitude higher than GUESS. Iterative deepening is also less efficient than GUESS, given that its control over extent is coarse-grained; however, we see even limited control of extent can result in a fairly good balance between cost and quality.

The reason fixed extent performs so poorly is because some queries are for popular items while others are for rare items, and the fixed extent can not adapt to these two extremes. For many queries that are for popular items, far more peers receive the query than is necessary to satisfy the query. However, if the fixed extent is made small, then the queries for rare items can not be satisfied. Having a flexible extent allows one to probe just as many peers as necessary. In the remainder of this section, we will focus on the challenges and opportunities afforded by a flexible extent: response time, and policy selection for efficiency.

Policy Performance. Here we examine the many options available for the various policies, with the goal of seeing which choices for a given policy are the most effective in the

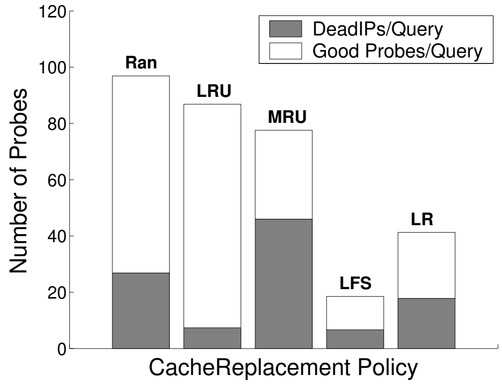

Figure 6.10: Probes/Query for different CacheReplacement policies

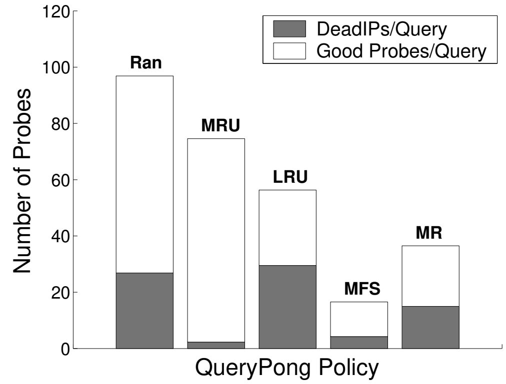

Figure 6.11: Probes/Query for different QueryPong policies

standard usage scenario (i.e., no capacity limits, no malicious behavior). We evaluate policies according to the number of probes used for queries and the percentage of queries that go unsatisfied. In terms of the number of probes used for queries, we distinguish between "useful" probes (those that get sent to live peers) and "wasted" probes which are sent to dead peers and therefore have no chance of returning useful information.

To simplify presentation of the results, in the rest of this section we fix the `PingProbe` and `PingPong` policies at the Random policy so that we can focus on query behavior. Also, in each of the graphs presented, the unspecified parameters use the default values in Table 6.1. In particular, aside from the policy being varied in the graph, the other policies are fixed as Random.

Looking at Figures 6.9, 6.10 and 6.11, our immediate observation is that choosing different policies can have a dramatic impact on performance. For example, in Figure 6.11 we see that changing the `QueryPong` policy can reduce cost (Probes/Query) by a factor of four. Likewise, in Figure 6.10 we see that changing the `CacheReplacement` policy can reduce cost by over a factor of five. The `QueryProbe` policy does not appear to make as significant a difference in performance compared to other policy types; changing the `QueryProbe` policy results in at most about a 25% change in cost. Certainly the `QueryProbe` policy should be chosen appropriately; however, as a first cut we recommend focusing attention on the other two policy types.

Another initial observation is that there are some policies with very serious drawbacks. The most obvious of these is the MRU policy for `CacheReplacement`; in Figure 6.10, we see that this policy causes a large number of probes to dead peers. This follows, since the policy evicts recently-contacted peers from its link cache, leaving the most stale entries. It appears that using MRU as a mechanism to enforce fairness does not result in effective search.

Looking at what does provide effective search, we see that the MFS `QueryPong` and LFS `CacheReplacement` policies are the most efficient (Figure 6.11 and Figure 6.10). In fact, used together, these policies result in query efficiency that is almost an order of magnitude better than if Random policies were used, thereby highlighting the importance of carefully selecting good policies. The MFS/LFS policies cause peers to circulate and maintain the identities of peers who share many files and are therefore more likely to return results to a query. The results-based policies (MR and LR) behave similarly, but are not quite as effective, because the number of results returned is not as accurate an indicator as number of files shared.

Unsatisfied Queries. Examining Figure 6.12, we observe that the proportion of unsatisfied queries is typically in the range of 6-14 percent. This figure may seem rather high given that one of the supposed advantages of GUESS is that it searches more extensively for rare files. This rate is partially an artifact of our simulation parameters; when simulating a network of only 1000 nodes, approximately 6% of the queries will go unsatisfied even if the entire network is probed, because some queries are for very rare or nonexistent items. In light of this effective lower bound on the unsatisfied query rate, we see that policies such as MFS and Random do quite well.

Response Time. Because each probe is performed sequentially, the response time of GUESS is linear in the number of probes required. Therefore, selecting a good policy is critical not only for efficiency, but for user experience as well.

To improve response time, a peer may send out k probes in parallel. Doing so will only increase the required number of probes by at most $k-1$, but will decrease response time by a factor of k – a good tradeoff for any moderate value of k (e.g., 10). For example, when `QueryPong = MFS`, the average number of probes required is 17. If we set $k=5$, and each set of parallel probes is sent out every .2 seconds (according to the GUESS specification [40]), then the average number of probes is at most 21, meaning average response time is less than 1 second.

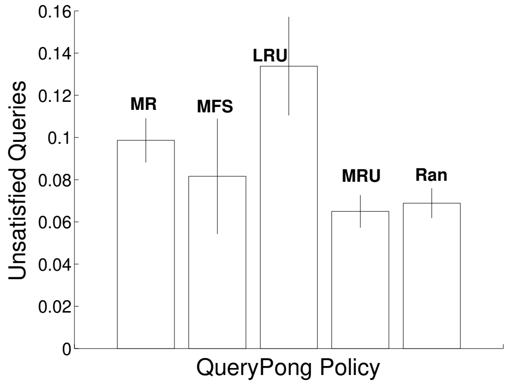

Figure 6.12: Percentage of queries that are not satisfied, for different `QueryPong` policies

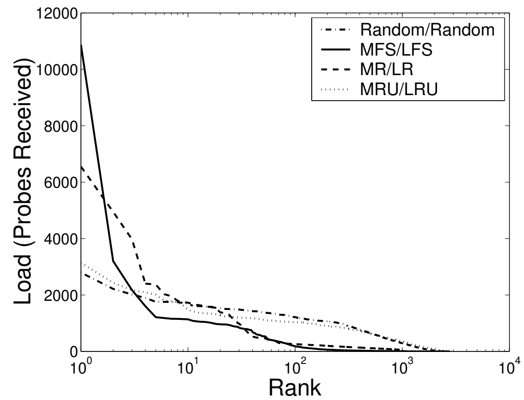

Figure 6.13: Ranked distribution of load (probes received) for different combinations of `QueryProbe` and `CacheReplacement` policies

Of course, though average response time may be low, worst-case response time is still bad. If 1000 probes are required to satisfy a query, then using the parameters in the previous example, 50 seconds are required to answer the query. A more sophisticated solution may adaptively increase k if successive sets of parallel probes are unsuccessful; we leave such a technique to future work.

6.5.3 Individual Loads

One of the main problems encountered by the original Gnutella protocol was congestion; therefore, it makes sense to examine whether GUESS might be susceptible to such problems as well. While GUESS does not cause queries to be magnified by a flooding mechanism (as in Gnutella), there may be other ways in which the limited capacities of peers come into play. We first investigate how different policies may lead to a high load for some peers, and then examine possible ways to remedy such a condition.

Fairness. Figure 6.13 shows the peers from a simulation run, ranked by number of probes received during their lifetimes, for different combinations of `QueryProbe` and `CacheReplacement` policies. The rank is shown on a logarithmic scale so that the highest-loaded peers are visible. We see that for MFS/LFS, and MR/LR, the load is weighted heavily to a small number of peers who do most of the work. On the other hand the curve for the Random/Random policy is much more level, meaning that the load is spread more evenly across all peers.

Despite the fact that we can choose a more fair policy, it is not clear that we want to. While the load is spread more evenly with the Random/Random policy, the total number of probes sent is over 8 times as many as when MFS/LFS is used. The probes are received in a more fair manner, but by peers who are unable to satisfy queries. This wastes both time and bandwidth for the querying peers as well as the receiving peers. So, fairness may be trumped by other concerns such as overall efficiency. However, this example does illustrate that some peers may encounter very high loads. Therefore, we should determine how the system will react if the capacity of these peers is exceeded.

Capacity Limits. In order to simulate peers having limited capacities we introduce the parameter `MaxProbesPerSecond`, representing the maximum number of probes that a peer can process within a one-second window. Any probes received beyond this limit will be dropped. In the following figures, we refer to these dropped probes as "refused" probes. Although different peers may have different capacities, we assume that in the interest of fairness, all peers set the same limit.

Under our default scenario, using a `NetworkSize` of 1000, a `CacheSize` of 100, and the default `QueryProbe` policy of Random, limited capacity barely comes into play. Any value of `MaxProbesPerSecond` greater than 1 results in less than 1 probe per query being refused (and in most cases, zero refused probes). Even with a capacity of just 1 probe per second, the number of unsatisfied queries increases by less than 2 percent. This is a promising sign that GUESS is resistant to congestion, which is one of the goals of the protocol.

We can see the effects of a limited capacity more clearly in larger networks, and in policies that are less fair, such as MFS and MR. Figure 6.14 shows the number of probes per query for different capacities and network sizes, using the MR policies. For example, the left-most group of bars in the figure shows the average number of probes per query for a network of 500 nodes, with capacity decreasing from left to right. As the network size grows, we see that when capacities are low, the number of refused probes is significant. For example, at a network size of 5000 and peer capacities of 1 probe per second, average number of probes per query is over 60% higher than when peer capacity is 50 probes per second, due to the increased number of refused probes.

Despite the increased number of refused probes, however, we observe that the number of good probes actually remains fairly constant as peer capacities vary. In addition, we see

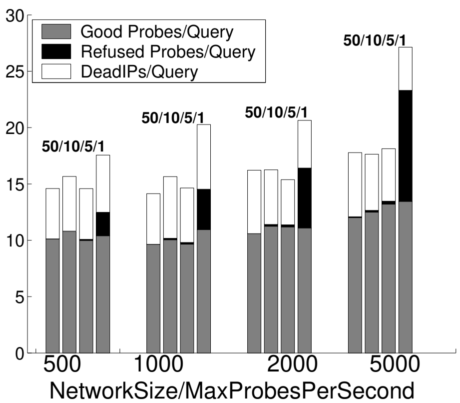

Figure 6.14: For large networks, limited capacity leads to more refused probes

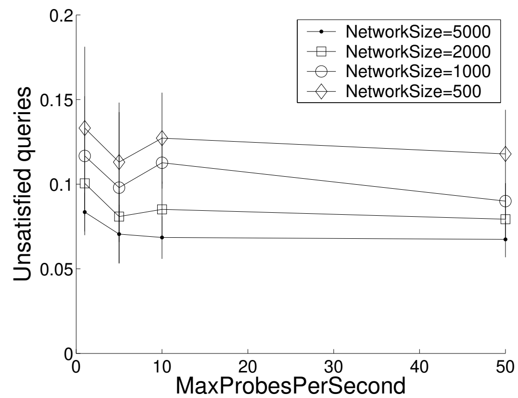

Figure 6.15: Query satisfaction is not affected by the capacity limits, even when a significant number of probes are refused

in Figure 6.15 that unsatisfaction rate remains constant as well. These observations suggest that if we could somehow predict which peers are overloaded and actively avoid probing them, then we could achieve roughly the same probes per query and unsatisfaction rate at 1 probe per second, as we can at 50 probes per second. In other words, the MR policy has the potential to be efficient *and* fair, given good heuristics to avoid overloaded peers. We leave the development of such heuristics as future work.

Nevertheless, GUESS cannot get around the fact that there are only a small number of peers that tend to share a large number of files. If a popular file is only located at a few overloaded nodes, many queries will still go unsatisfied. Ultimately, having a high degree of content replication is the best solution for allowing a file sharing network to scale successfully; it allows many querying peers to locate content without placing a huge load on any one or any handful of nodes. In a public trading network like Gnutella or GUESS, where it is difficult to require people to provide content, a better solution might be to provide incentives for sharing and replicating content.

6.5.4 Malicious Attacks

As with any wide-area P2P system, the existence of malicious peers trying to bring down the network is a possibility that must be considered. In most cases, the goal of a malicious peer is to make the system unusable. There are two main ways to accomplish this goal in a file-sharing network. First, the malicious peer may distribute corrupt or inauthentic files.

If many peers serve bad files, then good users may eventually become frustrated and leave. A second way to undermine the system is to attack weaknesses in the protocol itself. In Section 6.2 we already discuss how it is easy to flood peers in the GUESS protocol, and that some form of payment or score management is required to counter flooding. Another way to attack the GUESS protocol is to "poison" the link caches of good peers. As discussed in Section 6.5.1, if many entries in peers' link caches are dead, then query performance degrades. In the extreme case, the network becomes fragmented. If malicious peers return dead IP addresses, or addresses of other malicious peers, in their Pong messages, they may be able to inject enough bad entries into good peers' caches to bring down the network.

While the issue of document authenticity and reputation is important, it has been studied in various places (e.g., [20, 48]), and many techniques developed can be used on top of the GUESS protocol. Likewise, a fairly straightforward solution to flooding is available given the existence of an efficient score or payment mechanism. Hence, in this section, we study the robustness of policies to cache-poisoning. A policy is *robust* if query performance does not significantly degrade as the number of malicious peers in the system increases.

In addition to the five policies presented in Section 6.5.2, we also look at a sixth policy: MR*. As with MR, a peer P using MR* orders entries according to the number of results returned in the past, as specified by the `NumRes` field of the cache entry. Unlike MR, however, the MR* policy ignores the value of `NumRes` (resetting it to 0) if the field was set by a different peer. As a result, peer P will order entries based solely on P's direct experience with those peers, rather than the history of experience that other peers have had. While this policy causes potentially useful information to be lost, it also avoids making bad decisions based on corrupt information.

To poison good peers' caches, malicious peers will return either dead IP addresses in their Pong messages (`BadPongBehavior` = Dead), or they will return IP addresses of other malicious peers (`BadPongBehavior` = Bad). Note that in the second case, malicious peers must be colluding. When probed for a query, malicious peers return no query results; they will only return a corrupt Pong message. The fraction of the network that is malicious is given by `PercentBadPeers`.

No collusion. We first focus on the non-colluding case where `BadPongBehavior` = Dead. Figures 6.16 and 6.17 show us query performance in terms of average number of probes per query and unsatisfaction rate, respectively, as `PercentBadPeers` is varied. The different

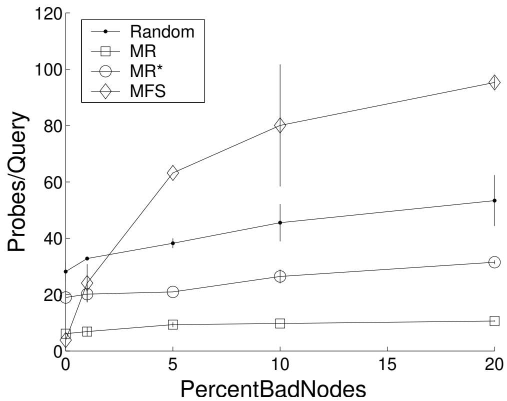

Figure 6.16: Average probes per query increases as the number of malicious peers increases

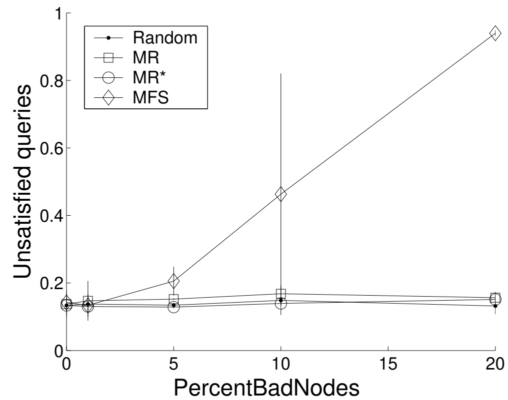

Figure 6.17: Unsatisfaction increases as the number of malicious peers increases

curves represent different combinations of policy types; the flatter the curve, the more robust the combination. For the experiments shown in these figures, we only vary `QueryProbe`, `QueryPong` and `CacheReplacement` policy types; furthermore, for simplicity, we assume that all three types implement the same policy (e.g., MR/MR/LR, or Ran/Ran/Ran). Each policy has the same qualitative effect regardless of the policy type it is used for; therefore the combinations of policy types we consider simply highlight the differences between the policies.

In these figures, we see that the Random, MR and MR* policies are robust against cache-poisoning with dead IP addresses, but that MFS is not robust. Although MFS is by far the best-performing policy when all peers are good (`PercentBadPeers` = 0), its performance quickly degrades, reaching a 0% satisfaction rate when 20% of all peers are malicious.

MFS is not robust because it requires that a peer fully trust other peers. If a peer P receives a pong entry reporting that another peer has a particular number of files, P will trust this information, and has no way of proving it wrong. Therefore, all dead IP addresses given by bad peers will be inserted into the link cache. In addition, since bad peers also purport to have many files, bad peers will also be inserted into and remain in the link cache. Though the dead IP addresses will be evicted after a single probe, the malicious peers remaining in the link cache will generate new dead IP addresses in their Pong messages, which will again be inserted into the good peer's link cache.

In Figure 6.18, we see the average number of good, or "unpoisoned," cache entries in

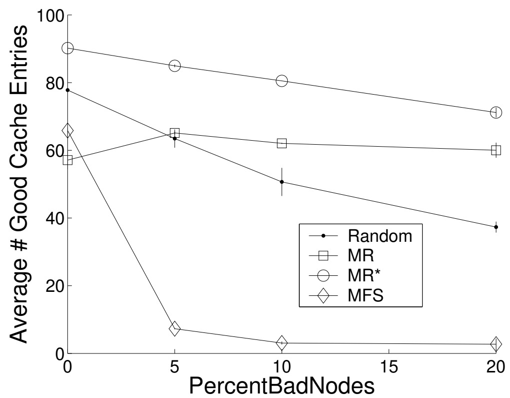

Figure 6.18: Good peers' link caches become highly poisoned as PercentBadPeers increases for MFS only

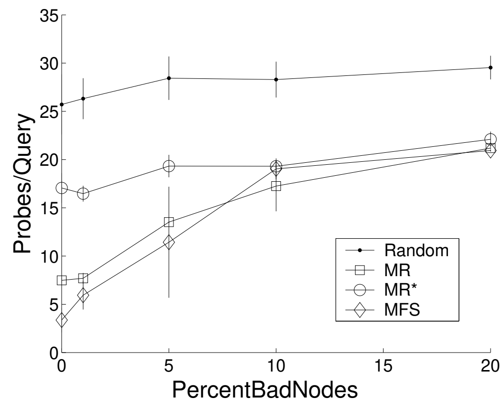

Figure 6.19: Average probes per query increases as the number of malicious peers increases

a good peers' link cache, as PercentBadPeers increases. We see that for the MFS policy only, the number of good entries drops off dramatically as the number of malicious peers grows. Because good peers no longer have any good entries in their link caches, they can no longer receive many quality results for their queries.

At first glance, MR should also have poor robustness similar to MFS. Because malicious peers report false NumRes values in the Pong message, all the dead IP addresses will be inserted into a good peer's link cache, just as with MFS. However, because the NumRes field is reset after each query, the malicious peers in the link cache will have their NumRes field set to 0, since they return no results, and will therefore likely be evicted within a short period of time. In contrast, recall that malicious peers remain in the link cache when MFS is used. Because of this difference, MR has excellent robustness when BadPongBehavior = Dead. As we see in Figure 6.18, the average number of "unpoisoned" cache entries in a good peers' link cache is barely affected by the number of malicious peers in the system.

The Random and MR* policies have good robustness as well. As expected, MR* has worse performance than MR, because MR* does not take advantage of other peers' experience. However, MR* outperforms Random because even a peer's limited, local experience with another peer is more useful than no information at all. Therefore, MR* will consistently outperform Random in terms of efficiency (number of probes), while maintaining the same effectiveness (satisfaction rate).

Because MR has the best performance by far for almost all values of PercentBadPeers

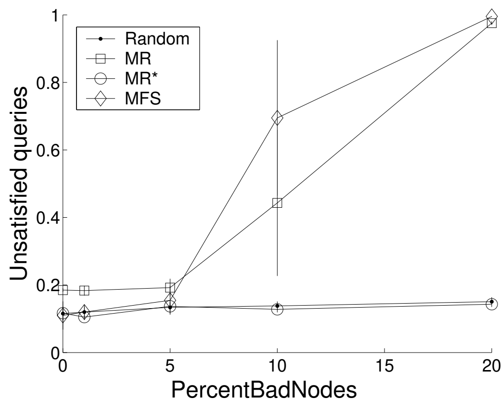

Figure 6.20: Satisfaction decreases as the number of malicious peers increases

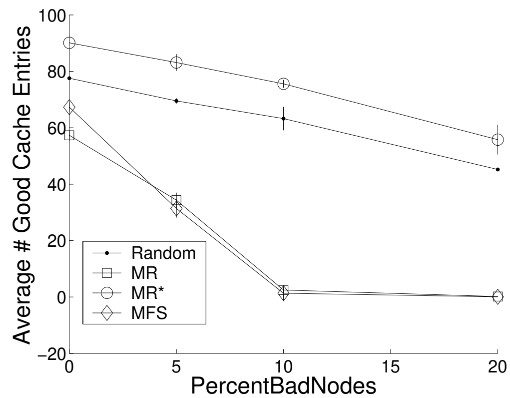

Figure 6.21: Good peers' link caches become highly poisoned as `PercentBadPeers` increases for MR and MFS

in the non-collusion scenario, it is the preferred strategy. However, we will see next that if malicious peers collude, MR is no longer robust.

Collusion. If malicious peers collude, they can return each others' IP addresses in their Pongs, rather than dead IP addresses. Doing so is much more harmful to the overall system, because whereas dead IP addresses are always evicted after a single probe, IP addresses of malicious peers are not. Malicious peers simply appear as good peers who happen to return no query results. In addition, when probed, malicious peers will return the IP addresses of other malicious peers, which will then be inserted into the good peer's link cache. In contrast, because dead peers do not respond, they can not further "poison" the good peer's cache.

In Figures 6.19 and 6.20, we see the robustness of the different policies as `PercentBadPeers` is varied. We set `BadPongBehavior` = Bad to model the case where malicious peers collude and return the addresses of other malicious peers in their Pongs. In these figures, we see that again, the MFS policy has the best performance when there are no malicious peers in the network. However, MFS is also not robust; when 20% of the network is malicious (`PercentBadPeers` = 20), 0% of the queries are satisfied (see Figure 6.20).

Likewise, MR has very poor robustness as well. Unlike in the non-colluding case, where `BadPongBehavior` = Dead, in these figures we see that both the average number of probes, and the unsatisfaction rate, increase rapidly as `PercentBadPeers` increases. Like MFS, when 20% of the network is malicious, MR yields 0% satisfied queries. This result may be surprising at first, because with MR, malicious peers are quickly evicted from good peers' link caches. Since malicious peers return no results, after one probe, the `NumRes` field for a malicious peer's is set to 0, making it a target for cache replacement. However, because probing a malicious peer once brings `PongSize` new malicious peers into the link cache, malicious peers enter the cache at a higher rate than at which they are evicted. Hence, even with the MR policy, good peers' caches remain highly "poisoned." We can corroborate this theory with Figure 6.21, which shows the average number of "unpoisoned" cache entries in a good peers' link cache as `PercentBadPeers` increases. As shown in this figure, good peers' link caches become highly poisoned (i.e., the number of good entries is small) as `PercentBadPeers` increases, for both the MR and MFS policies.

The Random and MR* policies once again exhibit good robustness, because they do not rely on information provided by other peers (number of files, number of results) to rank their cache entries. Furthermore, as before, MR* outperforms Random significantly, because it takes advantage of trusted, secure knowledge on the past performance of a given peer. However, although MR* is more robust than MR or MFS, when there are few malicious peers in the system, the performance of MR* is poor compared to that of MR and MFS. For example, at `PercentBadPeers` = 0, MR* requires 17 probes on average per query, while MFS requires just 4 probes, and MR requires 7.

In summary, the MR* policy is the best overall choice when malicious peers are a possibility. MR* outperforms Random consistently, but is much more robust than MR or MFS. Ideally, however, peers can learn to switch between MR and MR* if malicious peers are present. When few malicious peers are present, MR should be the policy of choice. If many malicious peers are present, the peer should use MR*. Although proving that a peer is malicious may be difficult, detecting malicious peers can be accomplished using heuristics – for example, if a group of peers constantly include each other in pongs, or if a peer consistently returns many dead IP addresses in its Pong. Hence, *detection* of malicious peers, and *adapting* policies based on this information, remains an interesting and important area of future work. The related issue of cache-poisoning prevention (without actually detecting bad peers) is addressed in [22].

6.6 Conclusion

In this chapter, we promote the concept of non-forwarding search mechanisms as a viable alternative to popular forwarding-based mechanisms such as Gnutella. Non-forwarding mechanisms, exemplified by the GUESS protocol, can achieve very efficient query performance, but must be carefully deployed. In particular, in this chapter we demonstrated how the *policies* used to determine the order of probes, pongs and cache replacement have a dramatic effect on performance and robustness. From our experiments, we concluded that the MR policy presents the best tradeoff between efficiency and robustness (Sections 6.5.2 and 6.5.4), while scaling fairly well with network size (Section 6.5.3). Therefore, our recommendation for a first-generation implementation of GUESS would be to use the MR policy.

Chapter 7

PPay: Efficient Micropayments

Although the "killer" application of free multimedia file-sharing propelled early P2P systems [35, 51, 62] into both the research and industry spotlight, it has recently come under legal fire from the recording industry. As a result, new P2P applications are emerging, such as pay-per-transfer file sharing systems, Grid-style computing systems, web service catalogs/service discovery and data storage/archival systems. The goal of these new applications is for the rightful owners of goods or services to be compensated; therefore these applications all share a common need: an efficient and secure *micropayment* mechanism, by which peers can purchase services from one another.

In addition, in the realm of resource discovery, the concept of goods and services extends to such actions as answering queries, forwarding messages, and so on. In an incentive-compatible resource discovery protocol such as that presented in Chapter 8, micropayments are likewise required to compensate peers for these services.

The problem with most existing micropayment schemes is the heavy load on the trusted, centralized broker. A broker is required to handle accounts, distribute and cash coins, and provide security (such as double-spending detection). Although payments need not be online, eventually the broker must take some action for every transaction; as a result, broker load is always $O(n)$ in the number of transactions. Brokers therefore present a scalability and performance bottleneck for any system using these micropayment schemes.

However, P2P applications have two main characteristics that we can exploit to address the scalability problem:

- First, peers serve as both *vendors*, who sell goods, and *buyers*, who purchase them.

As a result, a *transferable* coin can be used in many transactions before the broker must be involved in cashing it.

- Second, and more importantly, are the massive resources available at the peers themselves. Like other P2P applications, if we can tap into this pool of resources and shed the broker's load onto the peers, we can build a micropayment scheme with much better scalability and performance properties than existing ones.

The main challenge in exploiting the above two points is to ensure that the *security properties* of the scheme are not compromised. For example, transferable coins delay the detection of coin fraud, and we now want operations normally done by the trusted broker to be performed by untrusted peers.

In this chapter, we present PPay[1], a micropayment scheme that addresses the dual problem of improving performance while maintaining security. PPay only requires broker involvement when peers open or close accounts, for arbitration, and in limited cases, to perform services on behalf of offline peers. We show that under realistic application scenarios, PPay significantly outperforms existing schemes in terms of broker load. At the same time, however, PPay guarantees that all fraud is detectable and traceable. Our contributions are as follows:

- We present (Section 7.2) the basic PPay protocol for floating, self-managed coins.
- We show (Section 7.3) how the necessary security properties hold.
- We describe (Section 7.4) several extensions to the basic protocol that greatly improve system performance.
- We present a detailed performance analysis (Section 7.5), using simulations to (1) show PPay outperforms existing micropayment schemes in our target application scenario (Section 7.6.1), (2) determine the special cases in which PPay is *not* better, and (3) show how to tune the performance of PPay by setting the appropriate parameter values for our extensions (Section 7.6.2).

We believe that our performance analysis is unique in this field, where most schemes are proposed and only qualitatively analyzed.

Many of the contributions in this chapter were first published in [97].

[1] "PPay" is short for *PeerPay*

Preliminaries. One very important point to keep in mind is that micropayments are payments of a *small amount*. Therefore, (1) utmost security is not required, and (2) the payment mechanism must be lightweight, otherwise the cost of the scheme will outweigh the value of the payment. A real-world analogy to vending machines is illustrative. Because the goods are low-cost, it is not profitable to hire an employee to attend the machine. As a consequence, candy bars may be stolen, or clients may receive low-quality, stale candy. Likewise, most micropayment schemes (all schemes known to the authors) do not guarantee fair exchange of goods and payment. An escrow service (e.g., [43]) would make transactions more expensive than the value of the goods. Therefore, the main disincentive for cheating is that a bad user will be shunned, and will not be able to carry on business in the future.

Furthermore, because payments are "offline," meaning they are not processed by the broker in real time, coin fraud (e.g., using a counterfeit coin in a vending machine) may not be discovered until after the fact. However, offline payments are preferred from a practical standpoint because they have lower latency, communication costs and computational costs. Naturally, large-scale fraud needs to be detected and punished (discussed in Section 7.3). On the whole, however, effective micropayment systems simply need "good enough" security where fraud is detectable, traceable and unprofitable, while maintaining high efficiency.

7.1 Related Work

Many micropayment schemes have been proposed in the past, including NetBill [63], DigiCash [23], Millicent [34] and the scheme proposed by Rivest et. al. [58, 77]. Each of the above schemes require $O(n)$ broker load, where n is the total number of payments made. In fact, in DigiCash and NetBill, every transaction requires an online broker involvement, which as we noted earlier, has worse performance but better security.

Payword hash chains [77] and electronic lottery tickets [58] greatly reduce the "hidden constant" of this $O(n)$ cost in the work of Rivest et. al., but we note that both these techniques may also be applied directly to our scheme to reduce peer and broker load (see Sections 7.4.4 and 7.6.2).

The Millicent scheme also has a very low "hidden constant," because the broker does not need to perform any signature generations or verifications. As a result of this low security, however, there is no way of proving fraud. Instead, the broker will eject any vendor for which it receives many complaints. The money spent by the broker to pre-purchase that vendor's scrip will be lost. Furthermore, given the possibility of malicious peers "framing" a particular user, we prefer a system with stronger security properties.

At first glance, *transferable coins* [3, 9] seem to exploit the first characteristic of P2P systems we mentioned in the previous section. The basic idea behind transferable coins is similar to *layered coins* discussed in Section 7.4.2. When a user uses a coin for payment, he adds some information to the coin, such as his (possibly disguised/hidden) identity. Most transferable coin schemes have been developed in a macropayment context, and are very heavyweight in order to provide additional security properties such as anonymity and untraceability. In general, anonymity and untraceability are not used for micropayments because the cost outweighs the benefits. However, even a lightweight scheme employing the basic idea of transferable coins has the following properties: (1) the coin grows in size as it is transferred, and (2) double-spending detection is significantly delayed. For the above reasons, in practice, transferable coins must be cashed after they have been used a certain number of times. Even with transferable coins, then, broker load will still be $O(n)$ in the number of transactions. Furthermore, even if coins could be transferred indefinitely, eventually when the coins are cashed, the broker will receive $O(n)$ amount of data, because each coin's size is $O(n')$ in the number of times it was transferred.

Elements of the PPay protocol are *optimistic*, in that transactions between peers do not involve the broker unless an exception occurs (e.g., a peer dies). Optimistic protocols for fair exchange have been studied in the past (e.g., [4]). Unlike fair exchange, however, a micropayment protocol will always require some broker involvement in non-exception cases (e.g., when the final payment is made); hence, the focus of this work is to minimize broker involvement in non-exception cases.

Finally, MojoNation [60] developed a micropayment system for P2P applications; however, the company has since switched focus, and has left little documentation of their micropayment scheme. From informal interactions with others knowledgeable about MojoNation, it appears that their system used a technique similar to *soft credit windows* (described in Section 7.4.4), and had broker load that was $O(n)$ in the number of coin transfers.

In Section 7.6.1, we provide a performance comparison between our scheme and the micropayment scheme of [58, 77] in conjunction with transferable coins. For a survey over these and other schemes, please refer to [55].

7.2 PPay Protocol

In PPay, we present the concept of *floating, self-managed currency* to greatly reduce broker involvement. Floating currency allows digital currency to "float" from one node to another without the involvement of a centralized broker, which is only useful when peers act as both buyers and vendors. However, unlike traditional transferable cash, coins in PPay do not grow in size as they are transferred. We say the currency is "self-managed" because all security related to a coin, except for when the coin is first created or cashed, is managed by the users who "own" the coins. In this way, we harness the resources available at individual peers.

Notation. The public key of a node N is denoted PK_N, and the secret key SK_N. A message M signed by some key K is denoted as $\{M\}_K$ (e.g., $\{M\}_{SK_N}$).

Basic Implementation. Self-managed, floating currency is implemented as follows. A user U purchases digital coins from the broker B. A *raw coin* from the broker has the following form:

$$C = \{U, sn\}_{SK_B} \tag{7.1}$$

where sn is the *serial number* of a coin that uniquely identifies it. User U is now the *owner* of coin C.

Now, when U wishes to buy a service from another user V, it will send V the *assigned coin*:

$$A_{UV} = \{V, seq_1, C\}_{SK_U} \tag{7.2}$$

where seq_1 represents the *sequence number* of the assignment. Each assignment of a coin must have a sequence number that is greater than the sequence number of the previous assignment, if a previous assignment exists (that is, if this assignment is not the first of this coin). V is now the *holder* of the coin. At any time, a holder may "cash" its assigned coins (or any unused raw coins) at the broker for a flat fee (the fee is meant to deter users from constantly cashing coins, which reduces the benefits of floating currency).

Instead of cashing the assigned coin, V may instead pay another node X using the same coin. In the basic case, V sends the following *reassignment* request to owner U:

$$R_{UVX} = \{X, A_{UV}\}_{SK_V} \tag{7.3}$$

When U receives R_{UVX}, it keeps a record of the reassignment request in order to later prove that V "relinquished" that particular assignment of the coin, in the case of a dispute.

After receiving and processing the reassignment request, U then sends to V and X the new assignment:

$$A_{UX} = \{X, seq_2, C\}_{SK_U} \tag{7.4}$$

Nodes V and X are responsible for verifying that $seq_2 > seq_1$. After the new assignment has been issued, the old assignment, A_{UV}, is no longer valid. In Section 7.4.2 we discuss the option of *layered* coins, where V reassigns a coin to X without going through owner U. Note that both V and X receive the new assignment. That way, if for some reason X claims to have not received the coin, V can send its copy to X. If neither V nor X receive the coin, then the scenario is identical to the one in which U is simply down.

Downtime Protocol. Peers must be online to manage their coins. Say a node V receives a coin owned by U, and then attempts to spend it at node X. If U is unavailable (e.g., due to network partitions or natural downtime of a computer), or if U is attempting an attack, then it will not reassign the coin.

If the owner of a coin is not online (which the holder should determine via a ping *before* sending the reassignment request), the holder should attempt to make the payment via a coin owned by a different user who is online. In the case where this option is not available, or where the holder already sent the reassignment request but received no response, the holder may have the coin reassigned by the broker. Continuing our example from above, broker B will generate the newly assigned coin:

$$A_{UX} = \{X, seq_2, C\}_{SK_B} \tag{7.5}$$

which is identical in content to an assignment issued by owner U. The broker will then store the reassignment request from V and send it to owner U when it comes back online. U will be responsible for detecting any misconduct while it was down.

In the off chance where both the broker and the owner reassign the coin (e.g., because the holder timed out on waiting for a response from the owner), no damage is done. Both assignments will have the same sequence number and be assigned to the same user; hence they are essentially the same assignment. If the holder tries to reassign the coin to two different users, the owner can detect this misbehavior through the two reassignment requests. Please refer to Section 7.3 for a detailed security analysis of our protocol.

Broker B will charge both V and U a percentage of the reassigned amount for this service. Both parties must be charged in order to prevent attacks where U always pretends to be down, or V pretends U is down. Note that this extension provides incentives for nodes to remain online, and to minimize the involvement of the broker. It also makes any attack by the owner unprofitable, since the owner will not only be charged a fee, but it will also be unable to reuse the coin for itself.

An alternative to this downtime protocol is for peers to simply print a new coin whenever it does not hold another coin whose owner is alive. What a peer decides to do will depend on how much it is willing to manage another coin, and on what the respective fees are (which the broker controls). Later in Section 7.6.1 we present a performance analysis of this alternate method of dealing with offline owners. Even if this alternate method is used, however, we still need the downtime protocol to handle the case when the owner does not respond after the reassignment request has already been sent.

Member Certificates. Thus far, we have assumed peers know when other peers are a valid member of the system, and have knowledge of their public key. In practice, each peer is assigned a certificate upon opening an account with the broker. Like membership certificates described in [77], these certificates must have the following information:

$$cert = \{U, address_U, PK_U, expiry\}_{SK_B}$$

We also assume that all peers in the system have a way in which to obtain the public key of the broker over some authenticated channel. Notice that given a member certificate signed by the broker and the broker's public key, a peer can determine the public key of any other peer, and verify that the peer is a valid member of the system.

Whenever a peer V pays U by reassigning a coin owned by O, U must check O's member certificate under any one of the two following conditions:

1. U has never interacted with O before.

2. U has interacted with O in the past, but the last known certificate for O has expired.

In addition, in order to learn the public key of V (the peer making the payment), in order to verify the signature on the reassignment request, owner O must also check V's certificate under the same two conditions as above. If member certificates are not checked, then peers O or U open themselves up to attack.

We note that in *any* offline electronic check scheme, including PPay, a party receiving a payment needs to check the certificate of the party making the payment, or otherwise risk not getting paid in the end. Hence, for every payment in any generic lottery ticket scheme (such as the scheme in [77]), one certificate verification is needed under the same two conditions listed earlier. Also note that the conditions for certificate-checking imply that a peer caches information about other peers with whom it has recently interacted. Such a cache can greatly reduce the overhead of certificate-checking, especially given a stable user base. Given that peers have an incentive to stay with a system if they can make a profit, we do expect a reasonable amount of "membership stability" in a system using micropayments. In other words, although peers may go online and offline unpredictably, peers are not likely to close their accounts and permanently leave the system. Finally, we note that our focus is on broker load, and certificate-checking is purely client load; therefore, we do not further investigate how to reduce or avoid this overhead. We will discuss later (Section 7.6.1) why minimizing broker load is much more important than minimizing client load.

Discussion. We can make several observations about the general performance of the PPay micropayment protocol. First, under PPay, the broker's involvement is now limited to $O(m)$, where m is the number of coins issued. Since each coin can be involved in thousands of payments before being cashed, m may be orders of magnitude lower than n, the number of payments made. Furthermore, in the "ideal" case where all nodes own the same number of coins, and coins are randomly selected to make payments, the cost per node of handling coin reassignments remains constant, regardless of the number of nodes in the system – i.e., it scales perfectly.

Next, we observe that reassigning a coin is relatively expensive (2 signature generations, 4 signature verifications and 3 extra network messages). However, here we can directly use existing micropayment techniques originally intended to reduce broker involvement. For example, we can use *electronic lottery tickets* [58] to reduce the number of coin reassignments by a factor of s, where in practice s can safely be set to roughly 100. We refer readers to the citation for details.

Third, PPay is offline with respect to the broker, meaning that broker load is conserved. However, because payments must (usually) be online with respect to the coin owner, overall communication cost and latency is higher than with true offline payments. In Section 7.6.1 we compare the load of peers in PPay with other offline schemes.

Finally, we note that downtimes increase the involvement of the broker, but to a limited degree. As downtimes increase, PPay essentially degenerates into a regular brokered scheme where broker involvement is $O(n)$ in the number of transactions. In Section 7.6.1 we show how our scheme maintains good performance, even in the face of many offline peers.

7.3 Security

As with any offline payment system, *coin fraud* is an important issue that must be carefully addressed. Like other micropayment schemes, PPay does not prevent coin fraud at the outset, but instead makes fraud unprofitable. To achieve this goal, we need to ensure that (1) any fraud can be detected and traced back to the misbehaving peer (Section 7.3.1), and (2) proper punishments are in place (Section 7.3.2). Note that although a payment scheme can be designed to make fraud impossible (e.g., by making all payments online through the broker), such a scheme would be very inefficient. Recall that the goal of micropayment schemes is to provide an *efficient* payment mechanism with "good enough" security – i.e., where fraud is unprofitable.

Throughout this discussion, we assume that the broker is trustworthy. Broker fraud is unlikely for several reasons, as discussed in [34]. First and foremost, brokers will be businesses that rely on customer satisfaction and reputation for success. Even if a broker can commit fraud without being caught, its customers will realize they are losing money and will cease to use the broker's services. Second, because micropayments are worth very little, the broker will need to cheat a lot to gain any meaningful amount. However, large amounts of cheating will result in greater likelihood of being caught. Finally, by our protocol any broker misconduct can be detected and proven. In implementation, this detection mechanism will require that nodes keep a record of all coin assignments it has ever held; later we discuss how *coin renewals* (Section 7.4.3) can strictly limit the amount of state that must be kept. In practice, we do not feel that this detection mechanism will be necessary, due to the first reason. Hence, the following discussion will only focus on detecting and preventing peer misconduct.

7.3.1 Detection

To assure that coin owners and holders cannot commit fraud without leaving behind proof of misconduct, the following invariants must hold:

1. If H is the valid holder of coin C owned by U, then U does *not* have proof that H has

relinquished its valid assignment. Proof of relinquishment is in the form of a signed reassignment request from H for coin C, for the specific assignment in question (i.e., with the matching sequence number).

2. The sequence number of each assigned coin is greater than the sequence number of the preceding assignment of the coin.

3. If H is *not* the valid holder of coin C, then U can refute any assignment that H claims with proof of relinquishment (as defined earlier).

Note that as a consequence of invariant 3, the owner of a coin must keep a full "audit trail" of the coin.

To show that our system can detect and trace various attacks, we must prove the following theorem:

Theorem 1 *The above three invariants hold at all times, for all coins C.*

Proof of Theorem 1

To show that these invariants hold at all times, we begin with the base case, which begins with the issue of a new raw coin C and ends with the completion of the first reassignment of C. We then use induction to show that our invariants hold, or can be enforced, for each subsequent reassignment.

In the following analysis, we assume that public/private key encryption is secure, such that it is not possible for H' to forge an assignment with the owner's signature. As a result, for invariant 3, an "invalid" holder H' of coin C must have previously held C, and then relinquished it.

Base Case. First, the broker B issues a raw coin C to owner U. Invariants 1, 2 and 3 hold by default, because the coin has not yet been assigned.

Next, owner U assigns the coin to V via the following message:

$$A_{UV} = \{V, seq_1, ts, C\}_{SK_U} \tag{7.6}$$

Since V has not yet reassigned the coin, invariant 1 holds. Because A_{UV} is the first assignment of coin C, invariant 2 holds. Likewise, because no other node has previously held C, invariant 3 holds.

Say V decides to make a payment to X with coin C. V sends the following message to U:

$$R_{UVX} = \{X, ts, A_{UV}\}_{SK_V} \tag{7.7}$$

Here, we find one vulnerability of our scheme. At this point, invariant 1 no longer holds. Although V is still the valid holder of coin C until U issues the new assignment, U now has valid proof that V has relinquished the coin. Hence, U might never respond and thereby cheat V out of a coin.

However, note that from the perspective of all parties other than U, this attack is identical to the case where U is simply unavailable (i.e., not connected to the network). Therefore, no special effort is required to handle this attack; it can be addressed via the same exact mechanism used to address downtimes, discussed in the previous section. Furthermore, because U will be charged if it does not respond to a reassignment, and will not subsequently be able to use the coin for itself, U has no incentive to attack in this manner.

For now, let us assume U is available and not malicious. Invariant 2 holds because the current assignment (A_{UV}) is still the valid one. Invariant 3 holds because V is still the only holder in the history of coin C – hence, no invalid holders exist.

Now, say U responds to R_{UVX} with the new assignment:

$$A_{UX} = \{X, seq_2, ts, C\}_{SK_U} \tag{7.8}$$

At this point, X is the valid holder of C, and U does not have a reassignment from X; thus invariant 1 holds. Invariant 2 holds iff $seq_2 > seq_1$. If V or X detects that invariant 2 does not hold, then this assignment is not valid. However, V or X can present the invalid assignment, along with R_{UVX}, to the broker as proof of U's misconduct. Because U cannot produce a reassignment request for C to be assigned to X with a sequence number lower than seq_2, U will be punished (as described below). Therefore, U again has no incentive to attack in this manner, because it will be punished in addition to being forced to produce a valid assignment.

For now, let us assume that A_{UX} contains a valid sequence number. Invariant 3 also holds because V is the only previous holder of C, and U has R_{UVX} as proof of relinquishment from V.

Induction. Now, let us assume that the coin is reassigned n times, and that all three invariants hold. We will show that the invariants continue to hold after the $n+1$th reassignment. Say node Y is the nth holder of coin C, and that Y will now reassign coin C to node Z.

Case 1. Z has never held this coin before. Y sends the following request to owner U:

$$R_{UYZ} = \{Z, ts, A_{UY}\}_{SK_Y} \tag{7.9}$$

After the message is sent Y is still the valid holder until U responds with the newly assigned coin. Again, invariant 1 is temporarily broken, but can be dealt with as described above. At the end of the nth reassignment, invariants 2 and 3 were valid, according to our inductive hypothesis. Because the valid holder of the coin does not change as a result of this message, invariants 2 and 3 continue to hold.

When owner U responds with the new assignment:

$$A_{UZ} = \{Z, seq_n, ts, C\}_{SK_U} \tag{7.10}$$

then the valid holder of the coin changes from Y to Z. Invariant 1 holds because Z is the holder of the coin, and U has not received a reassignment request from Z. Invariant 2 may or may not hold initially, but given the mechanism described earlier to enforce correct behavior from owner U, eventually invariant 2 will hold. For invariant 3, by the inductive hypothesis, U has proof of relinquishment from the first $n-1$ holders of C. Y was the nth holder, and in the previous step, Y sent a reassignment request R_{UYZ} to U. U therefore has proof of relinquishment from the first n holders of C, which are all the previous holders of C – therefore, invariant 3 holds as well.

Case 2. Z has held this coin before. Y sends the following request to owner U:

$$R_{UYZ} = \{Y, ts, A_{UY}\}_{SK_Y} \tag{7.11}$$

All three invariants hold (or can be enforced) in the same way as in case 1. When owner U responds with the new assignment:

$$A_{UZ} = \{Z, seq_n, ts, C\}_{SK_U} \tag{7.12}$$

then again, invariants 2 and 3 hold, or can be enforced, in the same way as in case 1. Invariant 1 is slightly trickier, because U *does* have proof of relinquishment from Z for coin

C, but for a *previous assignment* of C to Z. Here, we see the importance of invariant 2: each assignment of a given coin has a unique sequence number, and a reassignment request (e.g., R_{UYZ}) contains the assigned coin (e.g., A_{UY}), which in turn contains the sequence number. Hence, owner U cannot use an old reassignment request to refute a new, valid assignment. As a result. invariant 1 holds as long as invariant 2 holds at all times.

Analysis of Downtime Protocol. The third invariant does not necessarily hold if the owner of a coin is down, and a holder goes to the broker for reassignment, because now the broker has proof of relinquishment of an assignment that the owner needs. Instead, we use a weaker (but sufficient) form of the invariant, which states that if H is *not* the valid holder holder of coin C, then an *omniscient party* O can refute any assignment that H claims with proof of relinquishment. By "omniscient party", we are referring to a (theoretical) party with access to all proofs of relinquishment relating to coin C, whether these proofs reside at the broker or owner, or both. If the broker is not storing any such proofs, the owner of the coin is an omniscient party. If the owner is currently down, and the broker reassigns coin C, then in practice, an omniscient party does not exist (but the invariant can still hold). Once the owner of C comes back online, however, it once again becomes an omniscient party, and can retroactively detect any misconduct regarding C.

In terms of state, the *downtime protocol* for reassignment (in which the broker reassigns the coin) is identical to the *default protocol* (in which the owner reassigns the coin) except that the proof of relinquishment is stored at the broker. Hence, at every step in the downtime protocol, an omniscient party O has the same state that owner U has at the analogous step in the default protocol. It is therefore clear that the weaker form of invariant 3 holds, by the same arguments used for the default protocol.

Attack Analysis

Given that Theorem 1 holds, it is now a simple matter for malicious peers to be detected and identified. Here we enumerate the most common attacks on a micropayment scheme, and show how all instances of the attack can be detected and traced back to the misbehaving peer.

First, the holder of a coin may commit fraud by **replicating** an assigned coin and spending it twice. For example, say owner U assigns a coin to V. User V then reassigns the coin to X, and then tries to reassign it again to Y. Given invariant 3, however, owner

U will always be able to refute the second reassignment of coin C. Note that if owner U was offline when V replicated its assignment, then the second, invalid assignment will be retroactively refuted when U comes back online, as described earlier.

The owner of a coin may commit fraud by wrongfully denying validity of an assignment, or by double-spending a coin (generating two coins with the same serial number). As an example of **wrongful denial**, a peer V may hold a valid assignment of coin C owned by peer U. If V tries to reassign C, and U claims that V's assignment is invalid, then V will lose the value of that coin. By our protocol, wrongful denial is prevented by default, because the burden falls upon the owner to prove that a given assignment is invalid. If an assignment is valid, by invariant 1, no such proof exists.

As an example of **double-spending**, an owner U may assign a raw coin C to V, and then assign the coin to X as well. The broker evades the double-spending attack by recording the serial number and owner of each coin that has been cashed, and the holder of the coin who cashed it. If multiple coins with the same owner and serial number are cashed, the broker will ask the owner of the coin to prove the invalidity of one or both of the assignments. If the owner is able to refute an assignment, then the holder of the coin will be punished. If the owner is *not* able to refute either assignment, then the owner will be punished. By invariants 1 and 3, the owner will only be punished if it double-spent the coin.

7.3.2 Punishment and Risk Management

In real life, if a business can be proven to be corrupt, then legal action can be taken. Likewise, because fraud can be exposed with proof, then the broker will be involved in the arbitration. In general, the broker B will require the misbehaving party to pay for any "fake" coins, as well as a penalty fee for misbehavior. If a node continues to misbehave, the broker may boot the node out of the system (e.g., never allow the node to cash in its coins, and/or inform other users to avoid the node).

Unfortunately, in any offline payment scheme, it is possible that when credit fraud is detected, the cheater is not able or available to make payments. In our system users must provide a valid credit card to the broker when they first join the system, and the broker can charge the penalty fees via this payment channel. However, if a user cheats, immediately closes down his credit card account, and disappears from the system, the broker can no longer charge the penalty fees. In the worst situation, the business owning the broker must settle with the user out-of-system, using the personal information provided when the user first joined the system (e.g., name, address, phone). Fortunately, we believe this type of "cheat and run" attack can be avoided in most or all cases given the following proper disincentives.

First, bear in mind that individual micropayments are worth very little. Hence, if the user cheats with a small number of coins, then the loss to the broker is negligible. To guard against medium to large cheat amounts, the broker may charge a deposit when the user first joins the system. When a user permanently leaves the system, he notifies the broker, who then recalls all coins belonging to that user. After the coins have been recalled and the broker has determined no cases of cheating, the user receives his deposit back, and may cash any coins it is currently holding. If the user cheats and disappears, without going through the formal process of leaving the system, he forfeits his deposit.

It is also possible that a user cheats with an enormous number of coins that a reasonable deposit cannot cover. However, such a scenario means that the user must purchase an enormous number of low-value goods (e.g., 100000 web pages), which is not likely or realistic behavior. This observation is in contrast to credit card fraud, where the cheater can select from a huge range of goods that are potentially very expensive. Furthermore, to select and download all these goods will take quite some time. Because the user cheated with an enormous number of coins, there is a greatly increased chance that the user will be caught before he finishes downloading the goods. Finally, the user will always be caught, eventually. Hence, even if the user is able to cheat and disappear before he is caught, he will eventually face a collection agency and bad credit as a consequence. Given these disincentives, then, we believe that large-scale cheating will be very rare or nonexistent.

In terms of a peer's risk, the broker will guarantee refunds to cheated peers in all cases except large-scale fraud. With large-scale fraud, the broker will provide refunds as far as can be covered by the bad peer's deposit. The remainder of the fraud amount will be covered if the broker can extract payment from the bad peer through out-of-system means. Individual peers must therefore manage their risk; e.g., by not accepting too many coins from the same user, or by periodically cashing their coins.

7.4 Issues and Extensions

Here we consider a few issues in the scheme described, and provide extensions that solve or mitigate the problems. For each extension proposed, we also discuss the associated security considerations and risks.

7.4.1 Limit Certificates

Printing raw coins can be expensive for the broker, given that the broker's load is otherwise very low. Observe that the purpose of a raw coin is to notify other users that the owner U has purchased this coin; hence, the coin can later be redeemed at the broker. To achieve this same functionality, we can instead have users print their own coins, given proof that the broker has authorized them to do so. When a user U first joins the network, it obtains a *limit certificate* from the broker that specifies how many coins U is authorized to print. The certificate has the following format:

$$L = \{U, lim_l, lim_u\}_{SK_B} \tag{7.13}$$

Field lim_l denotes the lower bound on the serial number of the coins that U may print, while lim_u denotes the upper bound. User U will typically need to prepay the value of $lim_u - lim_l + 1$ coins. U may now print his own "raw coin" with the following format:

$$C' = \{sn, L\}_{SK_U} \tag{7.14}$$

where it must be the case that $lim_l \leq sn \leq lim_u$. Any raw coin with a serial number outside of the limits set by the certificate is not valid.

Security Considerations. Note that while user U may print a coin with the same serial number twice, such an action is no different from U copying and double-spending a raw coin issued by the broker. The security properties of self-printed coins with limit certificates is therefore no different from the security properties of broker-printed raw coins.

7.4.2 Layered Coins

Rather than requiring a node V to go through owner U to reassign coin C to node X, V can instead reassign C itself by directly sending X the following message:

$$P_{UVX} = \{X, V, seq_n, A_{UV}\}_{SK_V} \tag{7.15}$$

where seq_n is greater than the sequence number of A_{UV}. Similarly, X can then reassign the coin to some node Y by adding another "layer" to P_{UVX}. Each layer of the coin is effectively a reassignment request, and serves as proof of relinquishment. When a node finally does go to U to have the coin reassigned, U can obtain all necessary proofs by "peeling" off the layers.

Layered coins are a lightweight implementation of traditional transferable cash, with no anonymity or untraceability properties. Like transferable cash, layered coins grow in size as they are transferred. Unlike transferable cash, however, layered coins in our scheme will eventually be peeled at a peer, rather than the broker; therefore broker load is significantly lower.

Security Considerations. In Appendix D.1, we show how Theorem 1 continues to hold in the presence of layers. However, layers may still delay the discovery of fraud. For example, say user V replicates coin C belonging to owner U. If layers are not allowed, V will be caught as soon as it tries to spend C the second time. With layers, however, V can reassign coin C to two users without U's knowledge. Only after both coins are peeled, can U pinpoint V as the misbehaving party. In Section 7.6.2 we study the tradeoff of layered coins between security and efficiency.

7.4.3 Coin renewal

To limit the amount of state each peer must keep (e.g., audit trails, payment history), we can require coins to be *renewed* during a certain time window (e.g., a 30-day window starting at time t). After the renewal window closes, the owner of the coin may purge the audit trail of the coin and start the audit trail anew. Further, any payment history involving an expired coin may also be purged. Any coin that has a renewal date in the past is considered invalid. If nodes do not renew their coins, then they forfeit the coins after the renewal window closes. Hence, the renewal window must be large enough such that nodes have a fair chance to renew the coins in their possession.

Just as with reassignment, if V tries to renew a coin at U but U is not alive, then V may have the coin renewed at broker B, at a fee for both V and U. Alternatively, V may simply cash the coin if it is about to expire, thereby taking it out of circulation. The decision V makes will depend on the respective fees charged for renewal and cashing.

Assigned coins now have the following form:

$$A_{UV} = \{V, ts, r_b, r_e, C\}_{SK_U} \tag{7.16}$$

where r_b and r_e designate the begin time and end time of the renewal period, respectively, and ts is the current timestamp. When V wants to renew a coin, it sends the following renewal request to U:

$$N = \{A_{UV}\}_{SK_V} \tag{7.17}$$

U will then return the assigned coin to V with a new renewal period specified:

$$A_{UV} = \{V, ts, r'_b, r'_e, C\}_{SK_U} \tag{7.18}$$

Security Considerations. In Appendix D.2, we discuss what happens if owner U tampers with the renewal window (e.g., sets the window to a time in the past) and how a holder V can easily guard against that attack. We also prove that Theorem 1 continues to hold in the presence of renewals. Here, we briefly discuss how the size of the renewal period affects the worst-case time to detect fraud.

As we mentioned in Section 7.3.2, we expect large-scale fraud to be rare, and likely to be detected within a short amount of time. However, in the worst case, fraud will not be detected until conflicting coins (i.e., with the same serial number) are cashed, reassigned or renewed at the broker. There are no guarantees as to when a given coin will be cashed or reassigned; however, the coin must be renewed within the renewal period. Hence, the worst-case time to detect any kind of fraud is equal to the renewal period. In practice, we must balance the tradeoff between renewal load and worst-case fraud detection. In Section 7.6.2, we discuss further this tradeoff between security and performance.

7.4.4 Soft Credit Windows

In some systems, payments are not only made for large items, such as downloads, but also for small items, such as forwarding messages or answering queries. Such systems use economic incentives to ensure the proper operation of the P2P resource discovery infrastructure. Because these "picopayments" may need to be very fast and will occur very frequently, we cannot require a signature generation and verification per payment. Furthermore, because the value of picopayments are very low (fraction of a micropayment), security surrounding them can also be low.

To make payments fast, we observe that in P2P systems, the economic relationship between nodes is often symmetric. For example, U may provide several services to V, but later, V will then provide services to U.[2] Due to this symmetry, payments between nodes can "cancel" each other out before the exchange of a signed lottery ticket or coin is required.

We take advantage of the above observation by utilizing *soft credit windows*.[3] For each node V that U interacts with, U must keep track of the balance M_{UV} of "picopayments" it owes that node. The balance begins at 0. Each time V performs a service for U, M_{UV} is increased. When U performs a service for V, M_{UV} is decreased. At the same time, V is maintaining M_{VU}. Under correct behavior, $M_{UV} = -M_{VU}$ at all times. Naturally, peers can easily behave incorrectly; we will discuss this problem shortly. When M_{UV} falls out of the range $(-x, x)$, then U must pay V, or vice versa. The balances are then reset to 0.

We make several observation about the performance of soft credit windows. First, if x is large enough to handle the variance in balance, theoretically, two nodes may never need to exchange coins, even if they provide enormous amounts of service for each other. Second, soft credit windows require you keep state per node you interact with, but this per-peer state is very small. Even if each node interacted with a million other peers in the system, total state would be on the order of 6MB of memory (i.e., one integer and one short per node), which is not at all significant on today's computers. Finally, soft credit windows are very efficient, not only because of the symmetric relationship between nodes, but also because no signature generation/verification is needed. Even if the relationship is totally asymmetric, we can still utilize soft credit windows to decrease the number of signature operations required.

Security Considerations. Because soft credit windows do not provide proof that one party is indebted to another, choosing window size x presents a tradeoff between efficiency and risk. As we observed earlier, large x may result in super-linear reduction in the number of coin reassignments. However, the larger x is, the more a peer stands to lose if it is cheated, for example, if the other peer refuses to make a payment when the balance reaches x. Note that it is possible for to U believe the balance has exceeded the window, but V

[2]Symmetric behavior may not be seen for large items, e.g., in download patterns. However, if payments are made for small items such as forwarding messages or processing queries, then relationships become much more symmetric.

[3]The authors have been informed that MojoNation [60] used a similar idea, which they called *bilateral accounting*. However, no documentation is available to confirm or cite. Beyond the introduction of this idea, we also discuss risk management for this technique, as well as a performance analysis in Section 7.6.2

disagrees. If this situation occurs frequently, then U should learn to avoid V. However, if the situation occurs infrequently (e.g., due to honest mistakes, lost messages), then the loss of a few picopayments is a very small deal. In Section 7.6.2 we will explore this tradeoff between efficiency and risk as x varies, and as the number of misbehaving peers in the system varies.

Because potential loss sustained by a peer can be high, we need to investigate ways of bounding this loss. We now present the *NetLossCap* method of managing the risk of credit loss.

Let $c_{ij}(t)$ denote the credit (in picopayments) peer p_j owes peer p_i at time t ($c_{ij}(t) = -c_{ji}(t)$). Potential *credit loss* $l_i(t)$ of peer p_i is defined as $l_i(t) = \sum_{p_j \in network} c_{ij}(t)$. The maximum value of $c_{ij}(t)$ is x (the size of the window), hence maximum possible loss $l_i(t)$ is $x \cdot \|network\|$. Loss is implicitly potential until a peer permanently leaves the system.

In the *NetLossCap* method, peers set a *cap value* v that limits credit loss l_i at any point in time. If $l_i(t) = v$ and a peer p_j wishes to extend its credit at p_i, p_i will refuse. Instead, in order to receive service, p_j will need to pay p_i a coin. In paying this coin, p_j is essentially extending p_i x picopayment credits (since a coin is worth one window's worth of picopayments). Loss $l_i(t)$ is thus decreased by x, and p_j can now receive service at p_i.

Note that when p_j pays p_i the coin, other requirements should not be broken. In particular, if $l_j(t) > v - x$ prior to the coin payment, then p_j cannot pay p_i – otherwise $l_j(t)$ will go over the cap value. In addition, p_j cannot pay p_i if $c_{ij}(t) > 0$ prior to the coin payment, otherwise $c_{ij}(t) + x$ will fall out of the credit window. No exchange of credit and service will occur unless p_j can afford to make the payment without unnecessarily affecting its own risk. In Section 7.6.2, we investigate the impact of *NetLossCap* on system performance and risk.

Hash Chains. An existing, popular way to make fast, small payments are Payword hash chains [77]. Hash chains seem perfect for picopayments because after an initial setup cost, each payment requires only a one-way hash. Two users A and B can set up a hash chain to denote A's payment to B, which requires a signature generation and verification. Each time A wishes to make a picopayment to B, it sends B the next "link" in the chain. A can verify this payment via a single hash. The chain has a pre-determined length, and once all links have been sent, A makes a real coin payment to B. Further details of hash chains can be found in [77].

Unlike soft credit windows, if the broker functionality is extended to honor partial

Payword chains, then Payword presents no risk. Payments can be proven by the links the hash chain, and partial chains can be redeemed when a peer permanently leaves the system.

However, using hash chains for a P2P application presents many practical issues. First, maintaining hash chains has high state overhead. Like soft credit windows, hash chains require peers to keep state per node they interact with. Unlike soft credit windows, however, this per-peer state is very large. Again let us assume that a peer interacts with a million other peers. If each hash "link" is 160 bits (the size of a SHA-1 digest), each chain consists of 100 links, and each peer has two hash chains per other peer (one to denote credit given, another to denote credit received), then total state will be an average of 4GB, which is *very* significant. Rather than storing full hash chains, peers can instead store only the base of the hash chain, and perform i hashes to reconstruct the ith link in the chain. However, doing so results in a processing load equivalent to one signature verification per picopayment, if a chain is 100 links long, which largely neutralizes the benefits of hash chains (see Section 7.5.3 for a discussion on calculating action costs). Although shorter hash chains can proportionally reduce state or processing cost, we will see later (Section 7.6.2) that they also result in significantly worse performance.

Therefore, hash chains are only really appropriate in applications where each peer only interacts with a small number of other peers. Even in this case, however, soft-credit windows can be much more efficient, due to the balanced relationships often observed between peers. Hash chains are a one-way payment mechanism that can only express an *increasing* amount of credit, rather than increasing or decreasing. Hence if users A and B exchange an infinite but balanced number of picopayments, they will need to exchange an infinite number of coins, rather than zero coins. In Section 7.6.2 we will investigate more closely the effect of this observation on the performance of hash chains relative to soft credit windows.

7.5 Experimental Setup

In the previous three sections we have provided a formal description of the PPay micropayment scheme. Now, we wish to use simulations to answer the following important questions: (1) Is PPay feasible in terms of broker load, peer load and the amount of state needed to implement our security features? (2) How does PPay perform relative to other schemes? What usage scenarios produce best or worst case behavior? (3) How should we tune parameter values (such as credit window size) for best performance?

Name	Default	Description
Network Size	1000	Number of peers in the network
Cache Size	100	Size of each peer's pong cache
Query Rate	$9.26 \cdot 10^{-3}$	The expected number of queries per user per second
Lifespan Multiplier	1	Multiplier used to extend the period of time peers remain online
Downtime Multiplier	1	Multiplier used to extend the period of time peers remain offline
Credit Window Size	10	Size of a soft credit window or hash chain
Credit Window Type	Payword	The type of credit window used (Payword, soft credit windows)
Max Layers	2	Maximum number of layers allowed on a coin before it must be peeled
Renewal Wait	30 days	Elapsed time between audit trail purges for a given coin
Scheme	PPay	The micropayment scheme used (PPay, PPay*, or EC)
Net Loss Cap	∞	Maximum allowed net loss per peer due to soft credit windows

Table 7.1: System and Micropayment parameters

We evaluate PPay in the context of a file-sharing system using the GUESS protocol [40] to discover files. As described in Chapter 6, one problem with the current GUESS protocol is that it does not provide the proper incentives for peers to cooperate – peers can easily cheat the system by sending out their probes all at once. This behavior incurs a much higher load on other peers, while drastically improving the response time of the search for the querying peer. If all peers act according to their best interests, the system might fail as if under a denial of service (DoS) attack. In order for GUESS to work, there must be a way to give peers an incentive to adhere to the protocol. One straightforward proposal is to have peers pay for each probe. Peers will then be motivated to probe as few peers as possible to answer their queries. In our evaluation, we use the PPay micropayment system to implement this incentive mechanism. In particular, peers will make one picopayment per probe. Coin payments are then made as credit windows are filled.

7.5.1 Metrics

Performance of a micropayment scheme is measured primarily by *load*, as described in Section 2.5. We look at load from two perspectives: the broker and the peers. Recall that our goal is not overall efficiency, but broker efficiency. At the same time, however, we want to ensure that peer loads are reasonable.

In addition to load, we also look at *state*, and *credit loss*. State is simply the number of "proofs" (i.e., reassignments or renewals) that an entity must store. Because the amount of state varies over time, we will be measuring the average state over time.

7.5.2 Parameters

We will be comparing the performance of different *configurations* of PPay, where a configuration is defined by a set of system and micropayment parameters, listed in Table 7.1. Most system parameters will remain unchanged in our experiments, as they do not qualitatively affect the results. We will describe these and other parameters in further detail as they appear later in the section. In our experiments, unless otherwise specified, the default parameter values are used.

7.5.3 Simulation Framework

In a given simulation run, we simulate `Network Size` peers participating in the GUESS protocol. At the beginning of the simulation, peers begin with a limit certificate but no printed coins (e.g., as if the peers just joined the network). Over time peers print new coins as they are needed. We assume the limit certificate is large enough to handle each peer's needs for the duration of the simulation, which is 40 simulated days. Throughout the simulation, peers may go offline and come back online. The duration of these online and offline periods follow the distribution of lifespans and downtimes measured by [82] over the Gnutella network. We may tune online or offline times via the `Lifespan Multiplier` and `Downtime Multiplier` parameters, respectively. If `Lifespan Multiplier` $= x$, then all values in the measured distribution of lifetimes are multiplied by x. Queries are simulated as described in Section 6.4 for the GUESS protocol. At the end of the simulation, we assume that all peers leave the system and cash in their coins. Doing so is a worst-case scenario for our scheme since in practice, the existing coins can continue to "float" before they are cashed.

When a peer makes a payment, it will first try to pay with a coin for which the owner is online. If the peer has no such coin, it will attempt to pay with a coin via layers. If all its coins already contain the maximum number of layers, the peer may reassign a coin through the broker, or print its own coin if it has a valid limit certificate (in Section 7.6.1 we compare these two alternatives). Within each class of coin (e.g., the class of coins whose owners are alive), the peer may often choose among several coins in its possession. The order in which coins are selected may have security and performance implications; for example, a peer V may choose a coin owned by another peer U, if V already holds many coins belonging to U. This policy decreases V's risk if U commits large-scale fraud. While the ordering of coins

Action	Bandwidth Cost (Bytes)	Processing Cost (Units)
Send/Recv Query	$70 + querylength$	1
Process Query	0	$14 + 1.1 \cdot$ # results
Send/Recv Response	$208 + 76 \cdot$ # results	1
Hash	0	5
Signature generation	0	5240
Signature verification	0	365
Send/Recv Coin	$172 + 44 \cdot$ # layers	$1 + .5 \cdot$ # layers
Send/Recv Reassignment	$200 + 44 \cdot$ # layers	$1 + .5 \cdot$ # layers
Send/Recv Renewal	$196 + 44 \cdot$ # layers	$1 + .5 \cdot$ # layers
Send/Recv Limit Cert.	190	1

Table 7.2: Costs of atomic actions

is an interesting question, in our simulations coins within a class are accessed through a queue.

Action Costs. Our simulations tell us how many actions each entity must execute. In order to calculate load, we now need to calculate to cost of these actions. There are several "macro" actions in our cost model: receiving and processing a query, reassigning a coin, requesting a coin renewal, etc. Each of these actions can be broken down into smaller "atomic actions" whose costs are listed in Table 7.2. These costs are calculated as described in Chapter 2, and the query-specific costs are taken directly from Table 2.3.

7.5.4 Note on Scalability

Our default network size of 1000 is a modest number of peers, compared to the scale of an actual P2P application. We had to limit network size because of the length of our simulations. However, most results shown in Section 7.6 are a good qualitative representation of any network size, because PPay is scalable.

Figure 7.1 shows broker load as the number of peers (parameter `Network Size`) is varied along the x-axis. Load is shown in terms of processing cost, but similar behavior is observed for bandwidth load as well. Similarly, Figure 7.2 shows peer processing load as `Network Size` is varied. All parameters in these experiments are set to default values, as given in Table 7.1.

For peer loads, although application-level purchases grows with the number of peers in the system, so does the number of peers that manage coins. In fact, peer load *decreases* as network size increases because more payments are absorbed by the greater total amount of picopayment credit (recall that maximum possible credit is equal to the network size

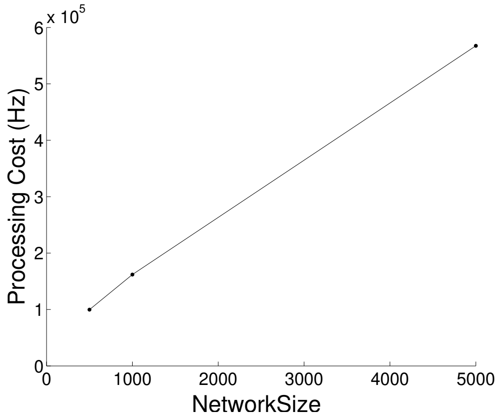

Figure 7.1: Broker load scales sublinearly as the number of peers in the system grows

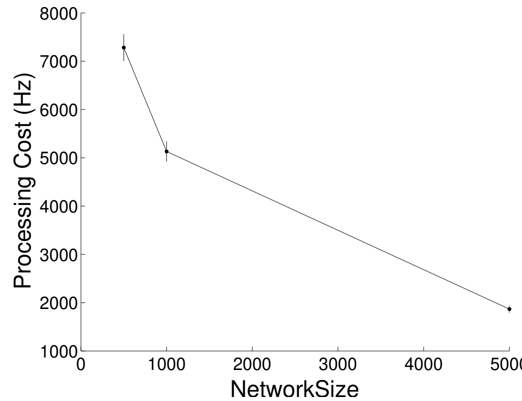

Figure 7.2: Peer load decreases as the number of peers in the system grows

times the size of the credit window). Broker load scales sublinearly with network size for the same reason. The only potentially unscalable result is the *NetLossCap* method for bounding credit loss, and we discuss this issue in Section 7.6.2.

7.6 Results

In this section we present and discuss the results of our simulations. We divide the results into two sections, corresponding to the two main questions that we wish to investigate: (1) the performance of the PPay micropayment scheme relative to other schemes, and (2) the tradeoffs involved in setting micropayment parameter values.

For clarity, we report all load results in terms of processing cost. Bandwidth results can be omitted because in all our experiments, either behavior is comparable to processing cost results, and/or the absolute bandwidth costs are very low, making them insignificant relative to processing costs.

7.6.1 Comparison with Existing Schemes

We compare PPay against the *electronic check* micropayment scheme that is used, for example, by Rivest et. al. in [58, 77]. We choose the electronic check (EC) scheme among many possible existing schemes because it is used in some of the most recent research in the area known to the authors, and because it provides a comparable level of security with PPay. In this scheme, users print tokens (like coins) to give to vendors as payment. Vendors then give these coins to the broker to redeem. In fact, the EC scheme is a special case of our scheme in which peers immediately[4] cash every coin they receive. On top of EC, Payword hash chains are used to delay the payment of a coin, as discussed in Section 7.4.4. We also allow the use of layered, transferable coins. The maximum number of layers allowed before a coin must be cashed is set by `Max Layers`.

In [58], Rivest et. al. also refine the idea of electronic lottery tickets, which reduce the number of coin payments without requiring large amount of state. Like Payword, electronic lottery tickets can be implemented on top of PPay as well. From the broker perspective, electronic lottery tickets have the same effect as Payword in reducing the number of coin payments by a constant factor. Hence, in our performance study we consider Payword, but not electronic lottery tickets.

We compare the EC scheme to two variants of PPay. By default, we assume peers prefer printing a new coin over using the downtime protocol. If a peer must make a payment, and it does not possess any coins whose owners are alive or that can be layered, the peer will print a new coin. In the second variant, which we call "PPay*", peers prefer using the downtime protocol over printing a new coin. Likewise for renewals, in the first variant peers prefer to cash a coin that is about to expire, rather than have it renewed through the broker; in PPay*, the preference is switched. While the first variant conserves broker load in terms of renewals and reassignments, it also causes more coins to be printed, which must eventually be cashed. In our performance studies, we want to see whether this tradeoff is worthwhile.

Broker Load. Figure 7.3 shows us the broker's load (work per unit time) in the three micropayment schemes in terms of processing cost, as `Downtime Multiplier` is varied. Bars are labeled below according to `Downtime Multiplier` value, and shade indicates the micropayment scheme. For now, let us focus on `Downtime Multiplier = 1`, the default case.

We see from Figure 7.3 that PPay and PPay* perform significantly better than EC. While the broker must cash $O(n)$ coins in EC, where n is the number of transactions, it must only cash $O(m)$ coins in PPay and PPay*, where m is the number of coins. Therefore, in this simulation EC broker load ($40.6 \cdot 10^5$ Hz) is over 12 times heavier than in PPay* ($3.46 \cdot 10^5$ Hz), and over 20 times heavier than in PPay ($1.98 \cdot 10^5$ Hz).

[4]By immediate, we mean that the coin is not reassigned to a different user before it is cashed.

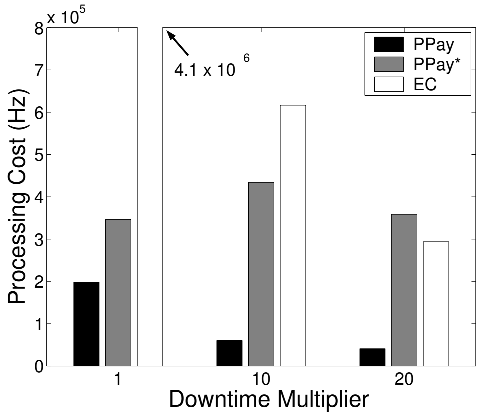

Figure 7.3: Comparison of broker processing loads in PPay vs. EC as Downtime Multiplier varies

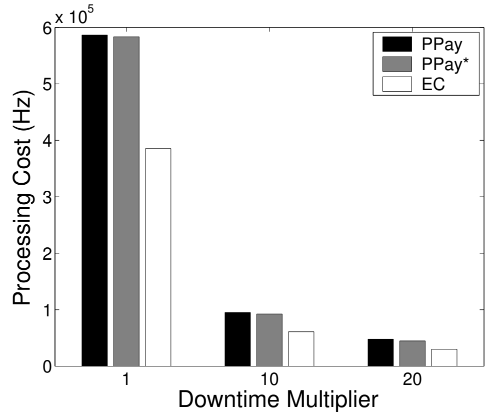

Figure 7.4: Comparison of peer processing loads in PPay vs. EC as Downtime Multiplier varies

At the same time, PPay outperforms PPay* because the broker no longer needs to reassign or renew any coins. Instead, for each reassignment/renewal the broker used to do in PPay* (which constituted about 43% of broker load), it now has to cash at most one new coin. Since cashing (1 verification) is substantially less expensive than reassignment or renewal (1 verification + 1 generation), broker load is likewise less expensive. The price of PPay falls onto the peer (discussed later), rather than the broker, which is a desirable tradeoff from our perspective. From Figure 7.3, then, we can conclude that floating, self-managed coins are crucial to minimizing broker load in a P2P application.

Recall that the distributions of online and offline periods are measured directly from the Gnutella network. Therefore, these distributions reflect the instability of peers, and their tendency to come online for short periods of time.[5] Despite peer instability, PPay and PPay* continue to perform well. One interesting question is how far can we "push" the instability of peers before the broker in PPay becomes overloaded.

Let us look again at Figure 7.3. In general, overall application-level activity decreases in absolute value as offline periods increase in duration (i.e., `Downtime Multiplier` increases), because there are fewer peers online performing queries. At `Downtime Multiplier` = 20, only 3% of all peers are online at any given time, on average. We believe this scenario is very extreme, yet PPay continues to far outperform EC. At `Downtime Multiplier` = 20, broker

[5]These distributions do *not* reflect the case where some peers may log on, log off, and then never come back to the network again. In our simulations, we assume that even though peers are online for short periods of time, they will always come back at some (possibly distant) time in the future. This assumption is reasonable in a system where peers have an incentive to come back (i.e., to make money).

load in EC ($2.94 \cdot 10^5$ Hz) is over 7 times heavier than load in PPay ($.41 \cdot 10^5$ Hz). However, broker load in PPay* actually *increases* as application-level activity decreases, because the number of reassignments and renewals that must go through the broker increases. At `Downtime Multiplier` = 20, 89% of broker load in PPay* ($3.59 \cdot 10^5$ MHz) consists of reassignment load. As a result, PPay* has the highest load of all.

Client Load. While we have achieved our goal of significantly reducing broker load, we still need to make sure that peer load does not suffer. Figure 7.4 is just like Figure 7.3, except that it shows average peer load, rather than broker load. In this figure, we see that peers in PPay do have higher load than peers in PPay*, but only slightly. As we mentioned earlier, peers in PPay will need to manage more coins than in PPay*. The cost of managing these coins comes from reassignments and renewals. However, note that since the number of reassignments is not affected by the number of coins in circulation, having more coins does not necessarily mean that reassignment load will be higher. Furthermore, renewals only constitute about 6% of total peer load in PPay, which is higher than it is in PPay*, but still not a significant cost. As a result, PPay has peer load that is comparable to PPay*.

Also in Figure 7.4, we see that peers in PPay and PPay* have processing load that is roughly 50% higher than peer loads in EC. Peer bandwidth load (not shown) is 25-35% heavier than in EC. While the increase in peer load is non-trivial, we believe it is a desirable tradeoff with broker load for two reasons. First, the absolute value of peer loads are small; users probably would not notice the extra load. Second, although the absolute values of broker load shown in these experiments are not much higher than peer loads, recall from Section 7.5.2 that peer load *decreases* slightly as the number of peers grows, while broker load *increases* (sublinearly). When network size grows, for example, to 1 million, broker load will be much higher than peer load. Therefore, from a scalability viewpoint, it is most important for broker load to be minimized.

Because PPay outperforms PPay* in all the scenarios we studied, in terms of broker load, we will only present results for PPay from this point on.

Worst Case for PPay. The relative performance of PPay degrades as coins are used fewer times before being cashed. Coins are used fewer times if owners are often offline, or the rate of buying and selling for a given peer is unbalanced (e.g., the peer spends more than it buys). In our simulations, a reasonable degree of imbalance and instability is already

Scheme	Bandwidth Cost (bps)	Processing Cost (Hz)
PPay(*)	$5.59 \cdot 10^3$	$3.82 \cdot 10^6$
EC	$3.75 \cdot 10^3$	$3.59 \cdot 10^6$

Table 7.3: Broker loads in EC and PPay, in the worst-case scenario for PPay

modeled: some peers have many more files than others (from 0 to over 70 thousand files), and some peers are online for much longer periods of time than others (from a few minutes to over 1000 hours). However, for argument's sake, let us consider the case in which coins are used exactly once before being cashed. Technically, in this scenario PPay degenerates exactly into the EC scheme. However, because coins in PPay include limit certificates (which EC does not need) they are larger.

Table 7.3 shows the cost of the two schemes when coins are used exactly once before being cashed. As we can see, broker load in terms of incoming and outgoing bandwidth is now 35% higher in PPay than in the EC scheme. However, in terms of processing cost, broker load in PPay is only 6.6% higher. Hence, even in the most extreme case, PPay performance is still comparable to the performance of EC. In the experiment described by Table 7.3, we note that electronic lottery tickets (an important aspect of the EC scheme of [58]), are not being used in either case, but can be applied to equally reduce the broker load in both.

However, in a complete comparison between EC and PPay, we also note that additional features are possible in EC due to the foreknowledge that coins will be "immediately" cashed. In particular, while electronic lottery tickets can be used in both schemes to equally reduce broker load, in the EC scheme, the broker can also implement the MR2 extension described in [58] to eliminate a buyer's risk of paying more than he should (for a discussion of client risk using electronic lottery tickets, we refer readers to the citation). This result is consistent with the fact that clients in PPay must manage more risk than in a completely centralized system.

7.6.2 Setting Micropayment Parameters

In the previous section, we used simulations to highlight the merits of PPay relative to other micropayment schemes. In this part of the performance study, we study how the performance of PPay can be tuned by varying the micropayment parameters listed in Table 7.1.

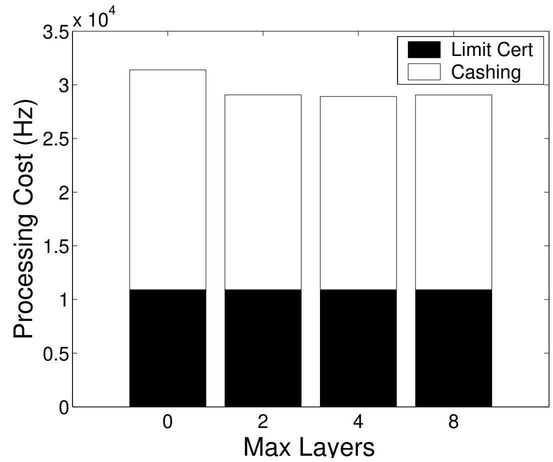

Figure 7.5: Broker load is barely affected by Max Layers

Layers

Figure 7.5 shows us broker processing load as the maximum number of allowed layers on a coin (`Max Layers`) is varied. In this figure, `Downtime Multiplier` = 20, because the more peers that are offline, the more impact layers will have on performance. As we can see, broker performance varies very little as `Max Layers` increases We initially found this result quite surprising. Because coin owners are often down when a payment must be made (with almost 97% probability, on average), we would have expected layers to be very useful in avoiding the need to print new coins. However, recall that new coins are printed in two payment scenarios: (1) when the peer has no coins, and (2) when a peer has coins but the owners are offline, and layers cannot be used (e.g., because the maximum number of layers has been reached). It turns out that printing more coins in scenario 2 results in printing fewer coins in scenario 1, such that the total number of coins printed is not largely affected by layers. Because layers have so little impact on broker performance (and peer performance), and since layers do delay the detection of fraud, we recommend that very low values for `Max Layers` (e.g., 0 or 1).

Coin Renewal

The `Renewal Wait` parameter specifies the system-wide renewal period for a coin. Renewals present a tradeoff between performance and security, and state management as well. As discussed in Section 7.4.3, the renewal period determines the worst-case time to fraud

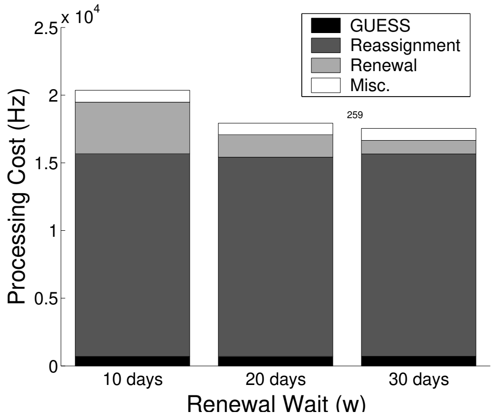

Figure 7.6: Peer load decreases as renewal period increases

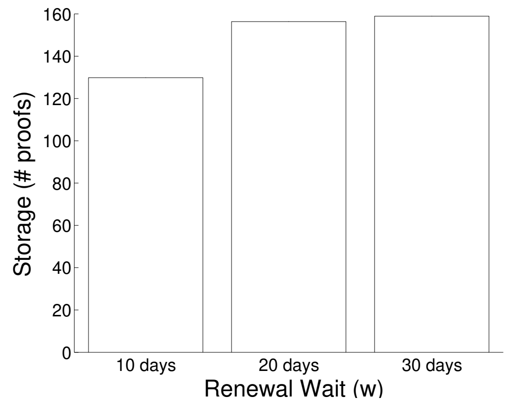

Figure 7.7: Peer storage (average number of proofs stored over time) increases as renewal period increases

detection. Therefore, the closer together renewals are, the tighter the bound. State also decreases, because audit trails are purged after each renewal period. However, load increases, since each renewal is fairly expensive.

In our implementation, we assume that that coins are renewed every `Renewal Wait` $=$ w days, and that the renewal window is also w days long. That is, if the renewal begin-time of an assignment is r_b, then after a renewal, the new renewal begin-time is $r_b + w$, and the renewal end-time is $r_b + 2 \cdot w$

Figures 7.6 and 7.7 illustrate the tradeoff between peer load and state as `Renewal Wait` is varied. Figure 7.6 shows peer processing load divided into four main tasks: GUESS-related work (e.g., processing queries), reassignments, renewals and miscellaneous (e.g., cashing coins). As we can see, renewals take up a non-trivial fraction of peer load, ranging from 20% at `Renewal Wait = 10 days` to 6% at `Renewal Wait = 30 days`. However, Figure 7.7 shows us that although `Renewal Wait` does have an impact on the amount of state a peer must store, this impact is limited. Therefore, the more important tradeoff is between performance and security. In terms of broker load, `Renewal Wait` has very little effect, since the number of coins printed due to payment is far higher than the number of coins printed due to expired coins. Therefore, since broker load is unaffected and peer load is scalable, we recommend a relatively low `Renewal Wait`, such as 10 days in our application scenario. Note that the renewal period should not be too low (e.g., 1 day), otherwise peers may not have a chance to renew their coins due to downtimes.

We note that in Figure 7.6, for peer loads, the overhead of PPay relative to the load of

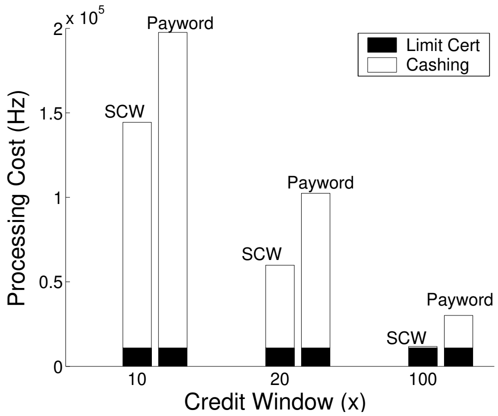

Figure 7.8: Soft credit windows have better broker load by accounting for symmetric peer relationships

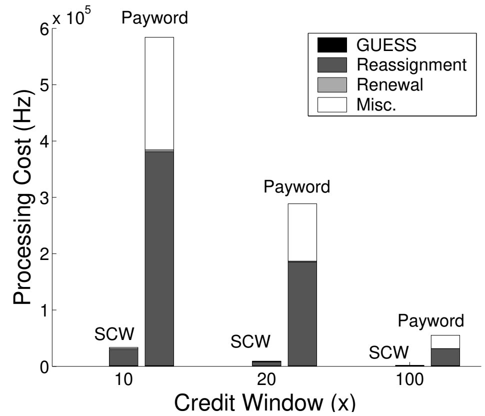

Figure 7.9: Soft credit windows have better client load by accounting for symmetric peer relationships

the application (GUESS) is very high - about 18 times higher. However, this comparison is skewed, as GUESS is not processing-intensive. In terms of bandwidth, the overhead of PPay is just 4% of the GUESS load. Furthermore, if we set `Credit Window Size` to 100 (default is 10), then even the processing overhead of PPay is just 13%, and the bandwidth overhead just .03%. In both processing and bandwidth cost, absolute overhead is always very low; therefore we do not believe the overhead of PPay for peers to be a significant issue.

Credit Windows

Figure 7.8 shows the processing load of the broker as `Credit Window Size` is varied, for both Payword hash chains (Payword) and soft credit windows (SCW). We immediately observe that credit window size greatly affects broker load – large credit window is crucial for good broker performance.

As we discussed in Section 7.4.4, Payword cannot take advantage of the "back and forth" symmetry of peer relationships. Soft credit windows, on the other hand, are designed to take this symmetry into account. In Figure 7.8, we see that soft credit windows consistently outperform Payword for this reason: broker load with Payword ranges from being over 40% heavier at `Credit Window Size`=10, to almost 3 times heavier at `Credit Window Size` = 100. Furthermore, peer load shows an even greater performance benefit with soft credit windows. Figure 7.9 shows peer load as a function of window size, in terms of processing

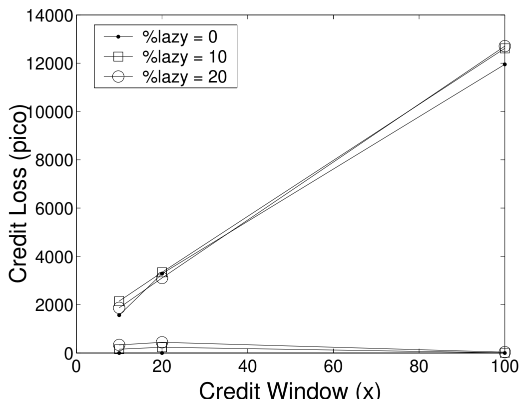

Figure 7.10: Maximum credit loss increases as credit window size increases

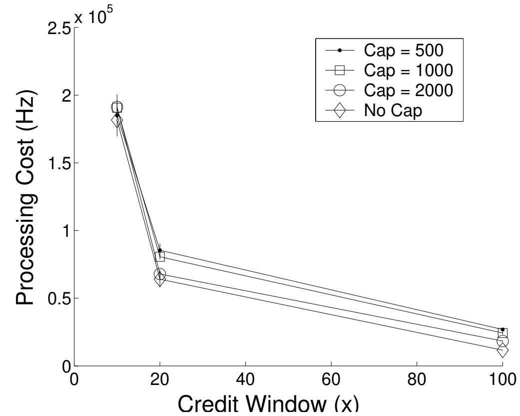

Figure 7.11: Broker load as the credit loss cap value varies

cost. Here, we see how Payword loads range from 17 times heavier (at a window size of 10) to over 38 times heavier (at a window size of 100)! Clearly, then, exploiting the symmetry of peer relationships is very beneficial to performance of both the broker and peers.

NetLossCap. Figures 7.8 and 7.9 show us the benefits of soft credit windows when no credit loss cap is applied. However, as we discussed in Section 7.4.4, in practice we need to bound the risk taken by peers using the *NetLossCap* method. Before discussing *NetLossCap*, let us define how a misbehaving peer might take advantage of soft credit windows.

We assume that the goal of a misbehaving peer is to receive the best quality service possible while spending no money. In other words, rather than being *malicious*, the peer is *lazy*. A lazy peer p_j will never pay a coin for service. Instead, it will only probe a peer p_i if $c_{ij} < x$, where x is the size of the credit window. Because a lazy peer p_j still wants to receive service, it will not lie about its view of c_{ij}, since that will cause peer p_i to shun it. Furthermore, the lazy peer is willing to provide service to other nodes, since it can potentially make money by doing so, and because it will at least receive credits to redeem for service in the future. For this same reason, then, the lazy node will not lie about its cap value (e.g., will not refuse service to another peer by claiming that $l_j = v$).

Now, let us see how effective *NetLossCap* is as `Credit Window Size` and `Percent Lazy Node` (the percentage of peers in the network that are lazy) vary. Recall that we assume all peers permanently leave the system at the end of the simulation. Credit loss for a peer p_i in the following figures therefore reflect $l_i(40 \text{ days})$.

First, Figure 7.10 shows us the mean and maximum credit losses of non-lazy peers in our simulation, when *NetLossCap* is not used. Along the x-axis we vary credit window size, while the different curves show different values of `Percent Lazy Node`. For example, when `Credit Window Size` = 100 when there are no lazy peers, the largest value of $l_i(40 \text{ days})$ observed for some peer p_i is roughly 12000 picopayments. This worst-case observed value is rare, however; 90% of all $l_i(40 \text{ days})$ values are under 4000, while the average value of $l_i(40 \text{ days})$ is 0.

We make several observations. First, the percentage of lazy peers in the system does not seem to greatly affect the risk of non-lazy peers. The reasons for this are that (1) credit risk is already very high, with no lazy peers, and (2) lazy peers are not cheating, per se – they are simply acting conservatively. Cheating peers will eventually be cut off from the rest of the system; hence, lazy/cheating peers do not pose a significant additional risk to the existing credit risk of an honest peer. Second, soft credit window size greatly affects potential credit loss. The need for a way to bound credit loss is clear, if we are going to reap the benefits of large credit windows.

When we apply the `NetLossCap` method with a cap value of v, the maximum observed credit loss is always v, regardless of credit window size. As an example, when window size is 100 and $v = 500$, maximum credit loss is 500, as opposed to 12000 when there is no cap. However, we need to be sure that using *NetLossCap* does not significantly affect performance – otherwise, hash chains may be a better alternative than soft credit windows.

Figure 7.11 shows us broker processing load as cap value and credit window size are varied. Window size is varied along the x-axis, and different curves represent different cap values. Although the lines may be difficult to distinguish, the basic shape of the curves tells us what we need to know. First, the choice of `Credit Window Size` is still very important in tuning performance, for all cap values shown. Second, assuming the cap value is not too small, cap values do not significantly degrade performance. Although this result may be surprising at first, the cause goes back to the law of large numbers. Even though credit between pairs of peers may be unbalanced (e.g., $c_{ij} = 100$ and $c_{ik} = -100$), the sum of these individual credits, l_i, will fall closer to the expected value, which is 0. Finally, we found (not shown here) that the quality of service received by each peer is not affected by *NetLossCap*, for the cap values studied. Hence, *in the given simulation scenario*, the *NetLossCap* method is very effective in limiting risk while maintaining good performance.

However, we note that the *NetLossCap* method is not necessarily scalable. As the number of peers in the system increase, if we assume that all users interact with all other peers, then the cap value must also increase proportionally. However, if the number of peers with which a given peer interacts does *not* grow with the size of the network, which may be reasonable in many situations (e.g., peers only interact with neighbors), then *NetLossCap does* scale. In either case, important future work lies in designing a quick payment mechanism that can take into account symmetric peer relationships, but can also provide proof of the credit balance to the broker.

7.7 Conclusion

Business-oriented P2P applications hold great potential for the future. Before such applications can take off, however, there must be an efficient way for peers to pay each other for services. In this chapter, we presented PPay, an efficient micropayment scheme designed for P2P applications. By identifying and exploiting characteristics unique to P2P applications, PPay can significantly outperform existing schemes in terms of broker load, while maintaining a reasonable peer load. We presented a detailed performance analysis over a sample P2P application to support our claims, and to illustrate how PPay can be tuned for best performance.

Chapter 8

RTR Protocol: Addressing Non-Cooperation

As we motivated in the previous chapter, increasing emphasis is being placed on new *legitimate*, economic applications of P2P, such as pay-per-transaction networks, P2P auctions, Grid-style computing systems and service discovery networks, where providers are compensated for their services. In these applications, peers *gain* from answering queries. For example, in a pay-per-transaction file-sharing network where peers get paid for uploading files, peers will want to share files, because this generates income. In an auction system, where the auction advertisement is analogous to a query, and bids analogous to query responses, peers will want to submit bids. Even in a free file-sharing network where users share their original music or artwork, users have an incentive to share their files to increase publicity. Therefore, not only are peers eager to provide services (e.g., share files, provide computational power), but they are in *competition* with other peers to provide their services.

Competition is a serious problem in P2P resource discovery frameworks that rely on peers to forward queries (e.g., Gnutella [35] and DHTs like [85]),[1] because a peer acting in its own best interests will not forward queries to potential competitors. For example, a peer providing a car rental service might not forward a query for car rental services. Instead, it could answer the query and then drop it, so as to improve its chances of gaining business. As a result, a *competitive network* in which nodes are in competition will no longer operate correctly due to non-cooperation, even though abundant services are available.

[1]The exceptions are systems that require no forwarding, such as hybrid systems.

In this chapter, we propose an economic protocol to ensure that peers cooperate in the *operation* of P2P networks in the face of competition. The basic idea behind the protocol, called the *RTR protocol*, is to have peers *purchase* queries from one another. Peers have both an incentive to buy queries, since they provide potential business, and an incentive to sell them, because they are assigned a price. In addition to overcoming competition, the RTR protocol improves overall system efficiency through techniques that are compatible with individual peer incentives.

We note that the work presented in this chapter is an early attempt at addressing a new problem. While much existing work has focused on performance of P2P systems, and others have looked at different types of incentives issues (e.g., free-riding in file-sharing systems, which is not applicable in competitive networks), ours is the first to address the dual issues of incentives *and* performance in resource discovery. Therefore, while we lay a foundation for addressing this new problem and identify the relevant issues, much of this work is still exploratory in nature. Further discussion of future work can be found in Chapter 9.

Our contributions in this chapter are as follows:

- We identify (Section 8.2) a new and relevant problem facing economic P2P applications, the *non-cooperation* problem, and quantify its effects.
- We present (Section 8.3) the *RTR protocol*, a potential solution to the non-cooperation problem.
- We discuss (Section 8.3.4) the robustness of this protocol, and provide means of avoiding and punishing cheating peers.
- We analyze (Section 8.5) the performance of the RTR protocol and show how it effectively addresses the non-cooperation problem.

In this work, we illustrate our protocol on top of the Gnutella protocol for P2P search running a pay-per-transaction file-sharing application. This application is chosen as a starting point for our study for two reasons. First, free file-sharing has proven itself to be a popular application, but due to its often illegal activities, a push is being made towards legal solutions involving payment (e.g., [46]). We therefore assume our users are ones who are willing to pay for file download, as are the users of [46], and to sell files under the approval of the copyrighted owners. Second, file-sharing has a naturally simple pricing model in which all files have the same value.

We selected the Gnutella protocol because, in the face of competition, unstructured networks are more appropriate than structured ones, which rely heavily on coordinated, cooperative behavior. Important future work lies in extending these ideas to systems with more complex valuation of goods and over different search architectures, such as DHTs (e.g., [85]). The following discussion assumes the existence of an efficient micropayment scheme for P2P systems, such as that described in [97], and a public-key infrastructure.

The basis of this chapter originally appear in [93].

8.1 Related Work

Researchers in ad-hoc and anonymity-preserving networks have also looked at providing economic incentives for peers to forward messages, such as in [8, 29, 100]. In each of these networks, peers are paid to forward messages. To ensure that peers are paid only as necessary, the Sprite [100] system uses a centralized server that processes a receipt of *every single* message that is forwarded in the network. In the the system proposed by [8], tamper-proof hardware is required at each peer to ensure proper payment is made for each forward. Finally, reference [29] requires that a peer sending a message knows exactly who will forward the message to its destination. Payments are then embedded in the message, and signed such that only the peer to whom the payment is made can redeem it. While each of these solutions are appropriate for their specific contexts, we cannot use them in the context of a general economic resource discovery system. For example, we cannot expect a peer trying to discover content to know a priori which peers will be needed to route the query. The RTR protocol differs from these approaches because it is designed specifically for use in *economic applications*. Hence, it can utilize the key fact that queries have *value* to peers.

There exist studies on incentives for many other aspects of P2P networks, such as sharing files (e.g, [36]) and answering queries (e.g., [86]). As we observed in Section 8.2, while many of these mechanisms can be modified to encourage message forwarding (e.g., to prevent *no-forward* behavior), they are unable to deter the subtle but harmful *competitive-forward* behavior. Because only a fraction of a peer's incoming messages are dropped under competitive-forward, these mechanisms cannot differentiate between peers who act competitively and peers who have, for example, one fewer neighbor than the average peer, as both will have a slightly lower output of queries.

8.2 Background and Motivation

Recall from Chapter 4 the basic Gnutella search protocol: each peer is connected to a small number of other peers (known as *neighbors*) in an overlay network. When a user submits a query, her peer will send the query message to all its neighbors, who will in turn forward the query to their neighbors, and so on. A peer that receives a query and finds that it can answer will send a response directly to the querying peer.[2] The querying peer will wait a period of time for responses to arrive, and then it will select one or more responding peers from which to buy services. In a pay-per-transaction application, the service offered is the download of a file, and the querying peer will pay the selected peer(s) for each download transaction. The price per download may vary depending on the file. Further details of the Gnutella protocol can be found in [35]. Clearly, if peers do not forward queries, the search mechanism will fail.

Effects of Non-Cooperation. In the case where peers cooperatively forward queries to each other, we say the peers are following the *all-forward* mode of behavior. In the non-cooperation case, peers can refuse to cooperate under two models: (1) *no-forward*, and (2) *competitive-forward*. Under the *no-forward* model, a peer forwards no queries. Technically, a peer under purely rational behavior will adopt no-forward behavior, because it would need to consume processing and bandwidth resources to forward queries, with no gain for itself.

However, no-forward behavior is easy to detect and punish; employing existing incentive mechanisms such as [86] may be enough to prevent peers from dropping all messages. A more subtle misbehavior is the *competitive-forward* model, in which peers do not forward queries if it will increase competition for that peer. To illustrate, consider a peer P in a file-sharing P2P application that responds to a query q. Because P will gain income if it is chosen to upload a file for q, it can maximize its expected income by not forwarding q to any of its neighbors, even if P only has an approximate answer (one that has just a small chance of being uploaded). Without a global view of the incoming and outgoing messages of a peer, which is unavailable in almost any P2P network, existing mechanisms are unable to detect competitive-forward behavior.

Under our two models of non-cooperation, we would like to see the effect of non-cooperation on performance. In Table 8.1, we compare the performance of the Gnutella

[2] In Gnutella, response messages are actually forwarded along the reverse path traveled by the query. We modify the protocol to be more efficient and respect the privacy of the responder.

	Score	# Peers Responded	# messages
All-forward	19.0	29.0	120.0
Competitive-forward	2.1	3.9	28.2
No-forward	1.1	1.9	4.0

Table 8.1: All, competitive and no-forward behavior

file-sharing network under the all, competitive and no-forward cases. A detailed description of the setup for this experiment can be found in Section 8.4. Here, we note that the experiment is over a sample network of 1000 peers, where each peer has a number of files across multiple categories. A query is for a specific file, but a peer will respond to the query if it contains any file in the same category. We measure performance in terms of quality of results and efficiency. Quality of results is reflected by how many peers responded to each query, on average, and the "score" of the response set (described in Section 8.4.3).

In Table 8.1, we see that both competitive and no-forward behaviors result in significantly degraded quality of results, when compared to all-forward. For example, in the no-forward case, the number of peers that process a query is 15 times smaller than in the all-forward case. Although competitive-forward outperforms no-forward, it still has an average response score that is just 12% of the score under all-forward behavior. Considering the importance of network variety to users of file-sharing networks, we see a clear need for overcoming non-cooperative behavior. We note that, not surprisingly, messaging overhead is roughly proportional to quality of results. However, since bandwidth and processing are renewable resources, we consider this cost small compared to the gain in user satisfaction.

Our goal in addressing non-cooperation is the following: *to allow the network to achieve the same quality of results and efficiency as all-forward, even in the face of competitive peers*. Indeed, we will see later that under the RTR protocol, the network adapts to form a more efficient topology over which we maximize the quality-to-cost ratio of the network.

Non-forwarding Protocols. Earlier we discussed why unstructured P2P networks such as Gnutella are more appropriate for non-cooperation than structured ones. However, one may also raise the question: why use any forwarding protocol at all? For example, why not use a protocol such as GUESS (Chapter 6), in which a peer sends its queries directly to other peers?

While a GUESS-like approach would address the problem of non-cooperation with regards to query-forwarding, we note that it introduces a new non-cooperation problem:

namely, that of maintaining peer lists via pong messages. Non-cooperating peers have no incentive to forward the identities or addresses of other peers in the network, because any peer could pose competition. As a result, a peer will know of only a few other peers in the network, and receive very few responses for its queries. Because it is not clear to us whether GUESS or Gnutella would be a better approach, and because Gnutella is much more widely used (GUESS has not yet been deployed), we choose to focus on a Gnutella-like protocol in this study.

8.3 RTR Protocol

At the core of our protocol is the concept of a *right to respond*, or RTR. An RTR is simply a token signifying that a peer has a right to respond to a query message. We choose this name ("right to respond") in order to emphasize that a query is really a commodity. Peers have an incentive to *pay* to receive the query, because that in turn brings in potential business. If a peer never receives any queries, then it can never provide its service to anyone. An analogous concept in real-life markets are companies that buy lists of emails or referrals from other companies, so that they have a new pool of potential customers.

Once a peer buys an RTR for a given query, it may do one or both of the following: (a) respond to the query and hope that it is chosen to upload its services, and (b) sell the RTR to other peers. Note that if a peer sells an RTR, it may still respond to the query corresponding to the RTR. That is, a peer does not lose the right to respond to a query when it sells the RTR for that query. Peers can buy and sell RTRs with their neighbors only.

In this framework, selling an RTR is equivalent to forwarding a query. There is clearly an incentive to forward queries, since peers earn income in doing so. Of course, some peers may still choose to not forward any queries in order to increase the probability that they will be chosen to provide the service. However, their actions will be offset by those peers who hedge their risk by selling a few RTRs, and by those peers who speculate in RTRs (buying RTRs simply to resell them).

It is important to note that all actions taken by the peer are *automated* – buying and selling RTRs, pricing, filtering and making connections (described later). While human users could theoretically make each decision, doing so would be tedious, error-prone, and not worth the time investment. Instead, the most reasonable implementation follows the model

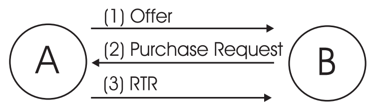

Figure 8.1: RTR Protocol

used by investment companies in the real world: users specify their high-level preferences, such as risk level, and the peer sets decision parameters to produce behavior accordingly. We will discuss these specific parameters in further detail later (Section 8.4.2).

8.3.1 Basic Implementation

An RTR has the following format:

$$RTR = \{Q, ts, query\}_{SK_Q} \tag{8.1}$$

where Q is the identity of the querying peer, ts is the timestamp at which the query was first issued, and $query$ is the actual query string. These three values are signed by the querying peer's secret key SK_Q, so that RTRs cannot be forged. Hence, each query requires a single signature generation, and optionally one verification per forward.[3]

When a peer A forwards a query to a neighbor B, it will first send an *offer* containing partial RTR information and a price:

$$\mathit{Offer} = \{rep(Q), ts, query, price\} \tag{8.2}$$

where $rep(Q)$ is the reputation of the querying peer (described below). Creating and sending an offer requires that peer A make several decisions. First, using quality and relevance filters discussed in further detail below, peer A must intelligently select which RTRs neighbor B will want to receive. Peer A must also determine a price at which to offer the RTR (discussed in the next section).

An offer contains enough information for B to determine whether to purchase the RTR, and whether the RTR is a duplicate B has seen before. However, because the identity of Q is not revealed, B cannot actually answer the query without purchasing the full RTR. If B decides not to purchase the RTR, B will simply drop the offer. Otherwise, B will send a *purchase request* to A, and peer A will forward the full RTR to B. The RTR protocol is summarized in Figure 8.1. After purchasing an RTR, a peer will respond directly to the

[3]Verification can be done on a random sampling basis to determine trustworthiness of a peer.

querying peer.

In terms of overhead, 3 messages are exchanged in the RTR protocol for every successful query forward, as opposed to 1 in the Gnutella protocol. However, we will see in Section 8.5 that the RTR protocol is just as efficient, sometimes *more* efficient, than Gnutella due to intelligent and incentive-compatible decision making.

Filters. To prevent being "spammed" by useless offers, each time a peer connects to a new neighbor, it can specify flow control parameters in the form of *filters*, that specify the "content" and "quality" of the desired RTRs. Filters can be set on any of the three fields of an RTR: the query string, the reputation of the querying peer $rep(Q)$, or the timestamp. Filters on the reputation of querying peer Q and the age of the query (indicated by the timestamp) specify the *quality* of the RTR, and affect the probability that a responding peer will be chosen and paid for its services.

Filters on the query string specify the *content* of the RTR, and affect the probability that the purchaser can respond to the query. Content filters may have varying levels of restrictiveness. For example, a content filter may specify that the RTRs should only contain queries for a particular genre of music files. Or, a content filter may specify exactly what RTRs (i.e., for which exact files) will be purchased.[4] At the other extreme, the filter can be null, meaning all queries qualify. Users simply select the level of specificity desired, and the peer will automatically set filters according to the files or services offered. For certain types of filters (e.g., categories or genres), we may assume the existence of a predefined taxonomy of services and files.

The semantics of content filters, if specified, are that they comprehensively cover the types of RTRs a peer may respond to, or wishes to resell. Thus, if a peer "spams" a neighbor with irrelevant RTRs, that neighbor may punish the peer by disconnecting from the peer. Or, if a peer never purchases RTRs that fit its filters, its neighbor may disconnect and direct time and resources to a different, more profitable customer. We discuss further the issue of automatically disconnecting from misbehaving neighbors in Section 8.3.3, and incentives for truthful filter reporting in Section 8.3.4.

[4] Indeed, such a filter results in a *super-peer* relationship in which a query is only forwarded from super-peer to client if the client can definitely answer the query.

8.3.2 Pricing

In this section, we discuss a simple pricing model for RTRs. Like the pricing of any commodity in the real world, the pricing of RTRs will involve the estimation of many parameters, as well as a user's disposition (e.g., risk-averse). The purpose of the model is not to provide a straightforward price for the RTR, but to help us understand the factors that influence price, and pinpoint the parameters that need to be estimated. In Section 8.4.2 we describe how peers may use the model to automatically price RTRs and estimate the necessary parameters, based on high-level user inputs.

Model. An RTR has value for two reasons:

- The peer holding the RTR may respond to the query and be selected to upload the file (for which she is paid).
- The peer holding the RTR may resell the RTR to some or all of its neighbors.

Let RTR_f denote an RTR corresponding to a query for file f. We assume any file f has a well-known price $price(f)$. Let N_A be the random variable denoting the income generated by the RTR for peer A, if A owns RTR_f. The income generated for A by holding RTR_f has two components:

$$N_A = S_A + R_A \tag{8.3}$$

where S_A is the random variable denoting the income gained for *selling a file* to the querying peer, and R_A is the random variable denoting the income gained from *reselling the RTR*. The value of RTR_f for peer A is simply $E(N_A)$, the expected value of N_A.

If A responds to the query and is selected to upload a file f, $S_A = price(f)$. Otherwise, $S_A = 0$. Let Q be the set of files that A owns that are possible responses to RTR_f ($|Q| > 1$ in the case of approximate matches). We assume that if A can respond to the query, then it will. Let $p_A(r)$ denote the probability that A's response r is picked for upload, given that A responded to the query with response r. The expected value of S_A is then:

$$E(S_A) = \sum_{r \in Q} p_A(r) \cdot price(r) \tag{8.4}$$

Let I_N be the indicator variable denoting whether neighbor N buys the RTR from A ($I_N = 1$), or not ($I_N = 0$). Note that the price at which the peer sells the RTR may be different from the price at which it bought the RTR, and it may differ on a per-neighbor basis. Assuming a peer N pays the expected value $E(RTR_f, N)$ for RTR_f, the income R_A generated by reselling the RTR is:

$$E(R_A) = \sum_{N \in nb} E(I_N | E(RTR_f, N)) \cdot E(RTR_f, N) \tag{8.5}$$

where nb denotes the set of neighbors of A.

Combining equations 8.3, 8.4 and 8.5, we can get the following formula for the cost of an RTR:

$$\begin{aligned} E(N_A) &= E(S_A) + E(R_A) \\ &= I_A \cdot p_A \cdot price(f) + \sum_{N \in nb} E(I_N | E(RTR_f, N)) \\ &\cdot \; E(RTR_f, N) \end{aligned} \tag{8.6}$$

Parameter Estimation. The probability that A will be picked to upload the response r, $p_A(r)$, can be estimated in two simple ways. First, A can learn p_A over time by remembering how often it responded to a query for the same content, and how often it was chosen to upload. Second, we note that p_A can be estimated as the inverse frequency of file f across peers. Peer A can get an estimate of the frequency of its files through sampling, or perhaps statistics offered through a third party.

Estimating how many times an RTR can be resold, and at what price, can be aided by filters. If A knows, via content filter, that neighbor N owns responses to the query, then A can guess with fair confidence that N will purchase the RTR. Furthermore, the lower bound for the RTR should be $E(S_N)$. If A knows that N is a speculator who buys many RTRs, then A may also assume that N will buy RTR_f, but at a wholesale price. If the RTR does not fit N's filters, A can assume N will not purchase the RTR.

Note that by setting content filters, a peer N is "giving away" information about its preferences that can make it a target for higher prices. In Section 8.3.4, we will discuss incentives for peers to report filters truthfully.

In Section 8.4, we will describe an actual implementation of the above pricing model, including one concrete approach to parameter estimation.

8.3.3 Peer Interaction

One of the main decisions a peer makes is with whom to interact, and in what manner. Certain neighbors may be more profitable than others due to many factors, such as trustworthiness, or an alignment of their demand with the peer's supply. To model how a peer makes these interaction decisions, we define the notion of a *profile*, and rules by which a peer uses profiles to guide their decisions.

Definition. The *profile* of a peer is defined on a pair-wise basis, and summarizes the interests of that peer. A profile of peer B compiled by peer A consists of the following information:

- Whether peer B has downloaded from peer A.
- Peer B's past queries.
- The *profitability* of B, if B is a neighbor.

To determine the profitability of neighbor B, peer A keeps track of how much money he has made through interactions with B over time, either through selling B RTRs, uploading files as a result of an RTR bought from B, or uploading files directly to B. Peer A also keeps track of how much money he has spent on B, namely, the cost of RTRs bought from B, as well as the cost of RTRs bought from other neighbors for the sake of reselling to B. Profitability is then the difference between the money made and spent with B. In addition, peer A may factor in the cost of resources consumed by that neighbor. For example, if B constantly spams A with RTRs that do not match A's filters, then the profitability of B decreases.

With the given definition, all profile information is a simple history of past interactions. In addition, we note that a peer A does not need to keep a profile of every other peer in the network. Instead, A must keep a profile of each of its neighbors, and then profiles of just a handful of other peers, for connection purposes (described next).

Usage. Profiles are used to determine with whom connections should be made and broken. If the profile of a neighbor shows it to be unprofitable over a period of time, the peer shall drop that neighbor. When a connection is broken, typically, a new neighbor is chosen. Our model for choosing a new neighbor is as follows: A peer P connects to peer Q if Q has submitted queries for files that P owns, with preference given to Q if Q has bought a file from P. The rationale behind this rule is that peer Q is interested in the type of content that P shares, therefore P can increase his chances of receiving relevant RTRs if he connects to Q. The flip side is that P and Q are more likely to be in competition with each other, and selling each other RTRs may hurt their own chances of being chosen for an upload; however, we will see later that the benefits outweigh the costs. Overhead of connecting and disconnecting is low, given that peers only evaluate the quality of their neighbors periodically.

In the remainder of this chapter, we will refer to the above connection strategies as the *adaptivity* model, since peers are adapting to the profitability of their surrounding peers. We note that in some cases, a peer is unable to select with whom to interact – e.g., in an ad-hoc network where interactions are only possible with peers that are geographically close. We study this scenario as well in Section 8.5.1.

8.3.4 Cheating Peers

A classic problem in game-theory is to incentivize peers to act truthfully. If a peer can gain by lying or deceiving others, then under the assumption of rational peer behavior, it will do so. In this section, we present an analysis of how peers might cheat under the current RTR protocol, and how to provide incentives against such cheating.

We point out the distinction between *rational* peers that cheat to maximize their utility, and *malicious* peers that desire to bring down the network. In this section we focus on rational but cheating behavior, which may not be as malignant to the system, but is likely to be much more widespread. Much existing work has studied techniques for countering malicious behavior (e.g., denial-of-service attacks [21], inauthentic file provision [48]).

Bogus RTRs

A *bogus RTR* is an RTR for which the originating peer does not intend to buy a file. Selfish peers generate and sell bogus RTRs in order to make a profit off of other peers who believe the RTR is genuine and may lead to an upload. A certain number of "bogus" RTRs are to be expected. For example, a user may submit several queries before deciding what to download, without the intent of making money off of the RTRs. In general, however, bogus RTRs are harmful to other peers, and are inefficient overall.

A simple way to remove some incentive for bogus RTR attacks is to require that RTRs are free when they are forwarded by their owner. For example, if a peer Q wishes to submit a query, it must send the RTR for that query to its neighbors for free. Its neighbors may then resell the RTR for profit. Because a peer cannot make a profit off its own RTRs, it has no incentive to launch the bogus RTR attack.

However, a *collusion* of peers may still profit off the bogus RTR attack: a peer Q may send a bogus RTR to its friend R for free, and R may resell the RTR at a profit, which it shares with Q. Collusions are less likely than individual misbehavior, due to the overhead (both in real life and in network costs); however, a well-designed system still needs a way to prevent bogus RTRs.

We use two techniques to combat bogus RTRs: adaptivity and *file buyer* reputation. Both of these techniques punish peers *over time* (i.e., in a repeated-game setting). Hence, in the following discussion, we assume that it is not easy for peers to create new identities. Such an assumption is reasonable in a system that deals with currency – peers' identities must be tied to a real-world entity before they can be trusted to make a payment.

Adaptivity. In a scenario in which peers may choose their neighbors for interaction, peers already have an incentive to submit valid RTRs. Recall in the adaptivity model from Section 8.3.3 that peers disconnect from neighbors that are unprofitable, and connect to other peers that have been shown to have common interests. Because peers issuing bogus RTRs are not profitable, as their RTRs will never yield profit for the buyer, other neighbors will eventually disconnect from these cheating peers. In addition, because a peer P gives connection preference to those peers that have actually downloaded from P in the past, and a cheating peer M rarely or never downloads, M is not likely to be chosen to connect to good peers like P. Therefore, peers M get pushed to the "edge" of the network with few neighbors.

At the edge of the network, a peer receives both fewer results for its real queries (if it has any), and fewer RTRs from other good peers that it may respond to (assuming it is also trying to make a profit by selling files). We will see in Section 8.5.3 that in the end, such "edge" peers do not make as much profit as good peers that remain in the core of the network.

File-Buyer Reputation. A peer's *file-buyer reputation* represents how consistent a peer is in sending out RTRs and then actually buying a file. We may use a reputation mechanism such as [48] to track reputations, and protect against malicious peers "inflating" each others' scores. Reputation is then used to affect two things: the estimated value of the peer's RTRs, and the likelihood the peer is chosen by other peers to *sell* files.

Under the RTR protocol, the expected value of an RTR is multiplied by the probability that the originator of the RTR will indeed buy the file it is searching for, which is estimated by the originator's file-buyer reputation. If a peer's file-buyer reputation is low enough, then the profit that peer can make from selling bogus RTRs will not be worth the effort. In addition, when a good peer P submits a query and selects a peer for download, it will bias its selection towards those peers with good file-buyer reputations. By reciprocating good faith, those peers that contribute to the economy by buying items will also profit in the end, which we will show in Section 8.5.3.

The file-buyer reputation of a peer is handled as follows: If peer Q downloads from P, then P knows that Q is a good file buyer; therefore, it will increase its opinion of Q by some amount s. Otherwise, P will decrease its opinion of Q by a small amount r. Over time, those peers that buy many files will have a good buyer reputation, and those who never buy files will have a poor buyer reputation.

In some cases, we may wish to differentiate between those peers that are new to the network, and have never had the opportunity yet to buy files, and those peers that frequently submit bogus queries. To do so, a peer P may decrease its opinion of Q by a small amount r each time it buys an RTR for a query submitted by Q. In this manner, the more frequently peer Q submits bogus queries, the lower its reputation will be, relative to a new peer.

Discussion. Both the file-buyer reputation and adaptivity techniques have strengths and weaknesses. The adaptivity technique is attractive because it is a natural result of local peer decisions that also improve the efficiency of the network. However, it does not protect against malicious peers – although selfish, *rational* peers are no longer able to make a profit off bogus RTRs, *malicious* peers who care only to destroy the network might still be able to flood peers within a small radius of themselves. In contrast, with explicit file-buyer reputations, it becomes an easy matter to automatically drop any message with low reputation, thereby removing the risk of flooding. However, managing file-buyer reputations requires an infrastructure whose operations can be quite expensive [48]. In this study, we focus on the effectiveness of these techniques in making the bogus RTR attack unprofitable. As future work, we intend to see how existing techniques to prevent flooding attacks (e.g., [21]), other than global reputations, can be applied on top of the RTR protocol.

Reporting Filters

We will see in Section 8.4.2 how specifying filters may make a peer a target for higher prices; hence, peers have an apparent incentive to not specify their filters. However, filters are desirable because, as we will show in Section 8.5.1, they can provide the basis for intelligent routing, improving the overall efficiency of the network. Here, we discuss how to motivate peers to uniformly adhere to a type of filter (e.g., exact filters, or no filters), and to report filters truthfully.

Adherence to Filter Type. Our first problem is to motivate peers to adhere to a single type of filter. For example, if the system operates under the assumption that peers report exact filters (i.e., exact information on the files they are sharing), a peer must not unilaterally decide to report more general category or genre filters, or perhaps report no filters at all. Fortunately, incentives to adhere to a standard filter type exist naturally, due to the weak equilibrium present in systems with mostly uniform behavior. Peers can be "initialized" to have a certain behavior if, for example, client software is *initially* distributed by a single source (which is typically the case in peer-to-peer systems). In order to produce a peer with deviant behavior and apply it successfully, a user will need to rewrite the client software, and then convince a large number of users to *simultaneously* switch to the new client. If only a few switch, or the switch is performed slowly over time, deviant nodes will suffer loss, due to the weak equilibrium. Because of this risk, and the difficulty of large-scale coordination, profitable deviant behavior is unlikely and easy to prevent.

Truthfully Reporting Filters. Given that peers adhere to a single type of filter, we need to give peers an incentive to report such filters *truthfully*. For example, because filters may make a peer a target for higher prices, that peer may have an incentive to conceal information, or falsely report content it does not actually own. The incentives for truthful reporting differ depending on the type of filter used. Let us consider simple types of filters, which are discussed in further detail in Section 8.4:

- *Category*: The list of categories describing files in a peer's library.
- *Exact Files*: The list of files in a peer's library.

Incentives to truthfully report category filters are much like incentives to adhere to a type of filter: they exist naturally due to the weak equilibrium present in the system.

Consider what happens if a peer untruthfully reports a category C_A for RTRs that it does not actually wish to purchase. Because the peer will rarely or never purchase these RTRs, its neighbor will disconnect under the adaptivity model (Section 8.3.3), and nothing is gained. If the peer untruthfully does *not* report a category for which it *does* wish to purchase RTRs (e.g., because revealing information makes it a target for higher prices), it will not receive any RTRs (and thus, earn no income) for the unreported categories, since other peers will respect the filters as flow control parameters. At the same time, prices for RTRs in reported categories remain the same. Thus, because all (or most) other peers respect and operate on the information provided by filters, a weak equilibrium forms, in which an individual's best strategy is to report these filters truthfully.

However, the "exact filters" case is more difficult to handle. Here, the dominant strategy may indeed be to lie. Consider a strategy in which a peer P reports a single file for each category it has. As we will discuss in Section 8.4, assume that a peer's neighbors will sell all "related" RTRs to that peer – that is, RTRs for the exact file reported, as well as any file in the same category. In such a scenario, peer P's filters will not preclude any of the RTRs that are relevant to it. Because P will receive most[5] of the RTRs it would have under truth-telling, at a lower price, its income increases.

Fortunately, we will see in Section 8.5.2 that in terms of overall behavior, category filters are just as effective as exact filters in network efficiency. Hence, from the protocol designer's perspective, we can safely eliminate exact filters from our design space. However, note that if exact filters were needed for some reason, we could put some mechanisms in place to discourage lying. As an example, we could have peers sign their responses to queries, and their filter reports. If a peer A notices discrepancies in the responses and reported filters from peer B – for example, if a response contains many files that are not reported in the filter specification – then B can shun peer A and share the "proof" with other nodes as well. We do not argue that such a mechanism with exact filters is preferable over a network using category or no filters; we only state that it is possible.

[5] When a peer specifies a file in a filter, its neighbors are more likely to sell it relevant RTRs, because they are more likely to buy such relevant RTRs at a higher price from their neighbors, given the high resell value.

8.4 Simulation Model

To evaluate our protocol, we simulate a P2P file-sharing application operating the RTR protocol. In this section we describe our simulator, which we then use to experimentally analyze the protocol in Section 8.5.

8.4.1 Query and Content Model

Each file has a unique identifier, as well as a *category*. One may think of a file's category as being, for example, the genre of music of the file (rock, pop, classical, etc.), or as the artist of the file. Queries are for specific files (i.e., they specify the unique identifier of a file). An *exact match* for a query is a response for the file specified in the query. An *approximate match* for a query is a response for a file that is in the same category as the specified file, but has a different identifier. Approximate matches have some small probability of being uploaded, so a peer will respond to a query with all exact and approximate matches found in its library.

When a peer submits a query, it will typically receive multiple responses, from which it must select one to download. We assume that a peer's probability of downloading any file is independent of what responses it receives, with the exception of the case where no responses are received. When a peer chooses to download, we assume each response has a relative likelihood of being selected. All exact matches have a relative likelihood, or "score," of s_e. All approximate matches have a relative likelihood of s_a. The peer then rolls a weighted die to determine which file will be selected. The probability $p_{choose}(r, M)$ that a given response r will be selected from a set of available responses M is:

$$p_{choose}(r, M) = p_d \cdot \frac{score(r)}{\sum_{r' \in M} score(r')} \tag{8.7}$$

where $score(r)$ is the score of match r (s_a or s_e), M is the set of all matches received for a query, and p_d is the probability that the peer will download any file (e.g., if the peer submits bogus queries, then $p_d < 1$). Only one file is selected per query.

8.4.2 Behavior Modeling

In this section, we describe how we model the behavior of a peer, incorporating such decisions as selecting filters, setting prices, etc. Throughout our discussion, we will pinpoint the variables that define a peer's behavior.

We will make some simplifying assumptions in the model below. For a first cut, let us assume that peers have a general idea of how "popular" each file or category is. Future work can be done on estimating popularity (e.g., through past experience). Popularity of a category c, $catpop_c$, is defined by the the probability that a randomly chosen file among all file instances in the network has the category c. Similarly, popularity of a file f, $filepop_f$, is the probability that a randomly chosen file is an instance of file f. Let us also assume that all files have the same price, and that each peer knows how many files his neighbor has, and roughly how many files N_{files} exist in the entire network.

Filters. Each peer maintains a set of filters that tells other peers what RTRs they are interested in. For simplicity, we pre-define three types of filters:

- *No filters*: No information is revealed.
- *Category*: The list of categories describing files in a peer's library.
- *Exact Files*: The list of files in a peer's library.

Recall that greater specificity results in higher prices, but also less traffic, and less obligation to buy RTRs that have a low expected yield. The variable f_{filter} is a per-peer variable that denotes which filter level a peer chooses. As we discussed in Section 8.3.4, peers can be given strong incentives to set their f_{filter} parameter to a particular system-wide value.

Buying and Selling RTRs. Three main decisions must be made when peer A wishes to sell an RTR to neighbor B. First, peer A must decide whether to sell the RTR to B. Second, if peer A does decide to sell the RTR, it must select a price. Finally, when peer B receives the offer, it must evaluate the value of the RTR to itself, and whether it wishes to purchase the RTR at the given price.

Function *SellRTR* (shown above) takes as input an RTR and a peer (presumably a neighbor of the calling peer), and returns TRUE if the calling peer should sell the RTR to this neighbor, and FALSE otherwise. The algorithm is very simple: if the RTR passes all content and quality filters, then sell the RTR. For simplicity, we assume a universal standard for RTR quality: all peers use the same thresholds for reputation and age. The function *matchesFilter* returns true iff the RTR matches the content of the neighbors at the filtering level chosen by the neighbor.

```
SellRTR(RTR r, Peer N)
if (not matchesFilter(r, N)) then
    return FALSE;
if (r.age < threshhold_age) then
    return FALSE;
if (r.reputation < threshhold_rep) then
    return FALSE;
return TRUE;

PriceRTR(RTR r, Peer N)
return uploadIncome(N, r) ·f_greed
```

Given an RTR offer, the *BuyRTR* function returns true if the peer should purchase the RTR represented in the offer. A peer should purchase an RTR if the price is below the *expected value*, estimated according to the model in Section 8.3.2. The term $p_{nodup}(r)$ represents the probability that a neighbor has not already seen RTR r, which depends on the age of the RTR. Variable $f_{bargain}$ reflects the "bargain hunting" tendencies of a peer. A peer will only buy an RTR at a price that is a factor of $f_{bargain}$ *lower* than the expected value. The higher the "greed level" specified by the user, the larger the value used for the $f_{bargain}$ parameter by the peer.

The function *uploadIncome*(Peer N, RTR r) calculates the expected income from uploading a file for a given peer and RTR (Equation 8.4). This value is simply the sum of the expected incomes for each response m in N's library to RTR r. Note that in order to estimate $p_{choose}(r, M)$, we must estimate M – how many other files the queryer will receive. We discuss the matter of estimation shortly.

Finally, the function *PriceRTR* returns the price for an RTR that the calling peer should set for a given neighbor. Again, the value of the RTR depends on the likelihood that the neighbor will be selected for download, and on the likelihood that the neighbor can resell the RTR to its own neighbors. The first factor can be estimated via a call to *uploadIncome*. The second factor we fold into an overall price-adjusting factor f_{greed}. Each peer sets f_{greed} according to how much they wish to "inflate" the price of an RTR – a "risky" action that may yield higher gains, but may also result in loss due to fewer sold RTRs. A peer may set $f_{greed} = 1$ conservatively, to price the RTR according to the known upload value, or $f_{greed} < 1$, to reflect the uncertainty in estimation, or $f_{greed} > 1$, to reflect the additional resell value of the RTR to its neighbor.

Parameter Estimation. Let us first consider estimating the M parameter in Equation 8.7.

BuyRTR(RTR r)
// First calculate $E(R_A)$ (Equation 8.5)
$E_R = 0$;
for each neighbor N **do**
    E_R += $SellRTR(r, N) \cdot PriceRTR(r, N) \cdot p_{nodup}(r)$
// **self** is a reference to the calling peer
$E_S = uploadIncome(\textbf{self}, r)$
value = $E_S + E_R$
return (value $> r.price \cdot f_{bargain}$)

uploadIncome(Peer N, RTR r)
// Calculates $E(S_A)$ (Equation 8.4)
Q = set of all matches in N's library
M = (estimated) set of all matches returned for r
$E_S = 0$;
for each match m in Q **do**
    E_S += $p_{choose}(m, M) \cdot price(m)$
return (E_S)

To do this, a peer must guess the fraction f_{return} of all matching file instances that will be returned to the queryer as a response. Clearly, the choice of f_{return} will be determined by the peer's particular behavior model (e.g., how conservative a peer is). With this fraction, the peer can estimate the number of exact matches the queryer receives for a query for file f, $n_{exact} = filepop_f \cdot f_{return} \cdot N_{files}$, and the number of approximate matches similarly as $n_{approx} = catpop_c \cdot f_{return} \cdot N_{files}$, where f has category c. The denominator of the fraction in Equation 8.7 is therefore:

$$\sum_{r' \in M} score(r') = n_{exact} \cdot s_e + n_{approx} \cdot s_a \tag{8.8}$$

Now let us consider estimating the number of exact and approximate matches a neighbor has (i.e., variable Q in *uploadIncome*) for a query on file f in category c. We use simple conditional probabilities. If no filters are specified, we assume the neighbor's content is a representative sample of the network. The expected number of exact matches he has is $filepop_f \cdot \#files$, and the expected number of approximate matches is $catpop_c \cdot \#files$, where "# files" is the number of files owned by the neighbor.

With category filters, we first estimate the number of files the neighbor owns in the file's category by assuming files in the neighbor's collection are distributed across categories according to the popularity of the category. Let us say we estimate n_c files in category c. Then the expected number of exact matches is $\frac{filepop_f}{catpop_c} \cdot n_c$ and the expected number of approximate matches is n_c.

Name	RTR_{nf}	RTR_{cf}	RTR_{ef}	Description
f_{filter}	1	2	3	Filter level (1 = no filter, 2 = category filter, 3 = file filter)
f_{greed}	.5	"	"	Price adjustment in *PriceRTR*. Accounts for resell value, and estimation uncertainty
$f_{bargain}$	1	"	"	"Inflation" of RTR value for bargain-hunting peers
$p_{nodup}[]$	[1,.6,.4,0...]	"	"	Probability that a neighbor has not yet seen this RTR
f_{return}	.1	"	"	The fraction of all possible results in the network that will be returned for a query

Table 8.2: Behavior Model Variables

Finally, with exact filters, we can determine exactly how many exact and approximate matches a neighbor has for the query.

In summary, each peer has a vector of variables that define its behavior in the context of the model provided above. We list these parameters in Table 8.2. Note that we express the p_{nodup} variable as an array, where $p_{nodup}[i]$ is the probability that a neighbor has not yet seen an RTR that has already been passed through i hops. In Table 8.2, we present three default configurations: RTR_{nf}, RTR_{cf} and RTR_{ef}.

8.4.3 Metrics

We use three main metrics to measure the *overall performance* characteristics of the network, each of which are averaged across all queries within a run.

Average Score/Query: The "score" of a query is a weighted sum of the scores of responses received for that query. Recall that the score of exact and approximate matches are s_e and s_a, respectively. By default, we set $s_e = 1$ and $s_a = .01$; however, unless otherwise noted, the relative values of this metric in our performance comparisons are insensitive to these values. Note that we use score to measure quality of results, as opposed to using the simple number of results or satisfaction metrics presented in Section 2.5. Unlike these other metrics, score provides a notion of the *relevance* of the results to the query, which is necessary to measure to evaluate the RTR protocol.

Average Peers Processed/Query: the average number of peers that process a query. A peer processes a query if it has an exact or approximate answer.

 # Messages/Query: the total number of offer, purchase request and RTR messages passed per query.

Sometimes, we may refer to the **quality-to-cost** ratio of a network, which we define to be the ratio of the average score per query to the average number of messages per query.

Individual peer performance is measured mainly by a peer's money **balance** at the end of a run. Peers begin the simulation with a fixed number of currency units, which are then used to buy RTRs, and are gained by selling RTRs and uploading files. Again, we assume the existence of an efficient P2P micropayment system, such as [97]. At the end of the simulation, those peers with higher balances have made a greater profit than those peers with lower balances. We note that it would be unfair to call a peer "unprofitable" if it buys files that it desires; therefore, we assume the funds used to *purchase files* come from a separate source.

8.5 Results

In our experiments, we attempt to answer three important questions regarding the RTR protocol: (1) In which scenarios is the RTR protocol most needed, and how useful is it in these scenarios? (2) Which configurations of the protocol (e.g., parameter values) are most beneficial to the network as a whole, or to individual peers? (3) How well can the RTR protocol withstand the bogus RTR attack? We address each question in the following sections.

Our experiments are run using the Query Cycle Simulator, described in [83]. To make our simulations tractable we limit the number of peers to 1000, but we have experimented with larger networks, and our results continue to hold. All figures in the following sections reflect results averaged over multiple runs. Unless explicitly shown, variance on results is small.

8.5.1 Overall Performance

First, let us revisit our motivating example in Section 8.2. Table 8.3 shows us the same example from before, but with results for the RTR_{ef} configuration as well. Recall from Section 8.2 that due to non-cooperation in an competitive environment, quality of results degraded dramatically compared to the cooperative all-forward scenario – by a factor of 8 under competitive-forward, and by a factor of 17 under no-forward. Our goal for the RTR protocol was to achieve the same high quality of results and messaging efficiency as seen when peers are not competitive – i.e., all-forward.

From Table 8.3, we see that the RTR protocol achieves its goal remarkably well. In

	Score	# processed	# messages
All	19.0	29.0	120.0
Comp.	2.1	3.9	28.2
No	1.1	1.9	4.0
RTR_{ef}	19.3	29.5	122.5

Table 8.3: A comparison of RTR protocol against all, competitive, and no-forward

particular, it achieves the same quality of results *and* messaging overhead as all-forward – a surprising result given the exchange of messages required by the RTR protocol. It is clear, then, that under a competitive environment, for the given example, the RTR protocol is necessary to ensure quality search results.

In this section we provide a more thorough investigation of the overall performance of the RTR protocol. In particular, our goal in this section is two-fold: (1) to illustrate, via experiments, the magnitude of the non-cooperation problem in different scenarios (if the magnitude is small, competition is not a problem), and (2) to show how RTRs can address non-cooperation.

We have identified four factors that affect the need for a solution to non-cooperation: (1) whether peers adapt (as described in Section 8.3.3) (2) the *interest overlap* between peers (without adaptivity), (3) the number of neighbors per peer, and (4) as mentioned earlier, whether peers forward *competitively*, or not at all. In this section, we will focus on the first three factors.

Adaptivity. Under our adaptivity model described in Section 8.3.3, not only are individual utilities maximized under the RTR protocol, but the topology also evolves into a more efficient one in which "clusters" are formed around common interests. Figures 8.2 and 8.3 show the performance of all, competitive and no-forward networks, as well as a network running the RTR protocol under our default configurations in Table 8.2. For each type of network, we show performance, measured by average score and messaging overhead, both with and without adaptivity. As expected, in the cooperative all-forward scenario, we see that adaptivity decreases overhead by over 24% (Figure 8.3), while increasing average score by over 69% (Figure 8.2).

However, the quality of results of competitive-forward drops sharply with adaptivity, to just 11% of the score under all-forward, and 30% of the score under competitive-forward in the non-adaptive scenario (Figure 8.2). Because adaptivity causes peers to be surrounded by

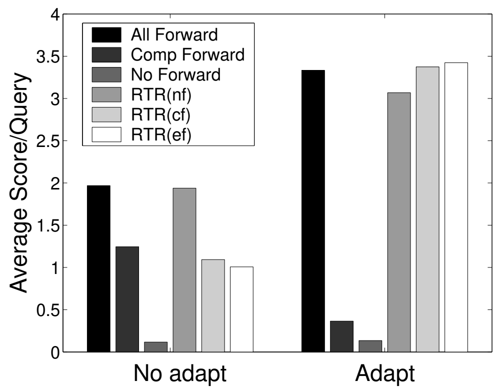

Figure 8.2: Competition becomes a greater problem with adaptive topologies

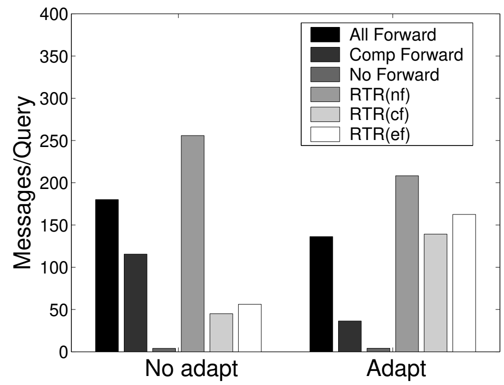

Figure 8.3: Messaging overhead decreases with adaptive topologies

other peers with similar interests, it only *exacerbates* the competition problem. Therefore, results in other adaptive topology studies (e.g., [13]) do not hold in the non-cooperative context.

Fortunately, under both the adaptive and non-adaptive scenarios, we can use the RTR protocol to maintain excellent quality of results at a very reasonable cost. In Figure 8.2, with no adaptivity, the RTR_{nf} configuration achieves 100% of the score possible under all-forward, at a 42% increase in messaging overhead. Such an overhead is reasonable given that for every peer that actually buys the query, three messages must be exchanged instead of one. Furthermore, notice that the RTR_{ef} and RTR_{cf} configurations, while not achieving as good quality of results, have a better quality-cost ratio than the all and competitive-forward cases. RTR_{cf} achieves almost 56% of the score of competitive-forward, at just 25% of the messaging cost. Under the adaptive scenario, the strengths of the RTR protocol become even more apparent. For example, the RTR_{cf} configuration actually achieves better quality of results than all-forward, at roughly the same cost.

Breakdown of Messaging Overhead. The RTR protocol can achieve such good quality-cost ratios, despite the need for 3 messages per bought query, due to efficient routing of queries to those peers who can provide answers. Figure 8.4 shows us the breakdown of message types for three configurations: all-forward, RTR_{nf} and RTR_{ef}. We show all configurations with and without adaptivity. Messages are divided into four categories: offer messages, purchase request messages from peers that have answers to the query, purchase request messages from peers without answers, and RTR messages (see Figure 8.1). Under

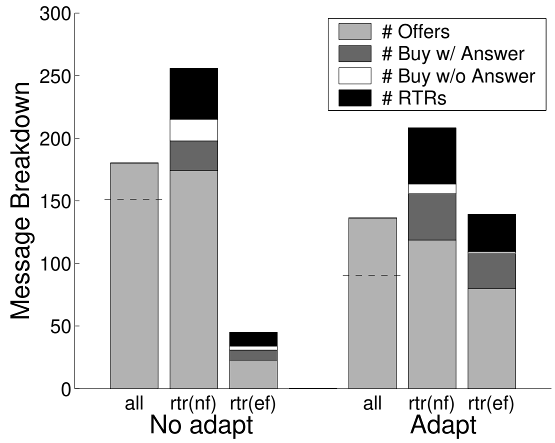

Figure 8.4: Breakdown of messaging overhead

the all-forward configuration, there is only one type of message – queries. In the figure, for all-forward, we draw a dotted line such that the portion *above* the line denotes useful queries (i.e., sent to a peer for the first time, and the peer can answer the query).

From Figure 8.4, we make three important observations. First, looking at all-forward, a large number of query messages are not useful. For example, in the no-adaptivity case, less than one sixth of all query messages are useful. Useful queries are the only messages that contribute to the quality of results returned for queries; therefore, if the RTR protocol can intelligently route queries only to peers that can answer them, then the protocol can decrease cost while maintaining high quality.

Second, recall that each query that is purchased results in an additional overhead of 2 messages per query. Since most purchased queries are useful ones (i.e., the purchaser can answer the query), and since the relative number of useful queries is low, the overhead of the RTR protocol is also low. For example, consider the RTR_{nf} configuration without adaptivity in Figure 8.4. Roughly 1 out of 4 queries (or offer messages) are purchased, so additional overhead is about 42% the original cost of forwarding queries, rather than 200%.

Finally, and most importantly, we observe that the RTR protocol can indeed reduce the fraction of query messages that are useless. For example, consider the RTR_{ef} configuration with no adaptivity. Roughly 1 out of 2 query messages are useful, compared to 1 out of 6 under all-forward. The difference between RTR_{nf} and RTR_{ef} is that RTR_{nf} uses no filters, and RTR_{ef} uses exact filters. Hence, we see that the presence of filters allows us to intelligently route messages.

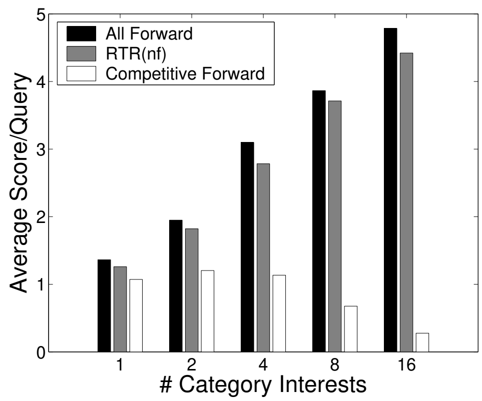

Figure 8.5: Competition becomes a greater problem when interest overlap increases

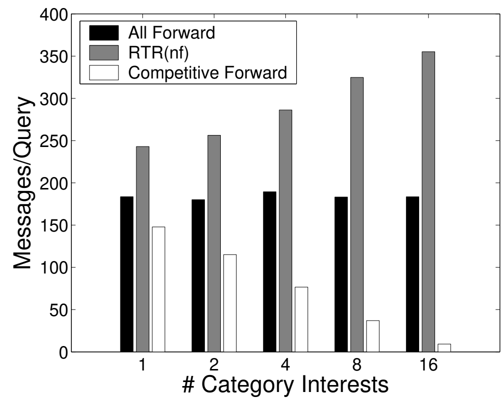

Figure 8.6: Messaging overhead

The problem with RTR_{ef} without adaptivity is that the absolute number of useful query messages is *less* than under all-forward or RTR_{nf}, thus resulting in worse quality of results. Under RTR_{ef} query messages can be prematurely dropped if none of a peer's neighbors have filters that match the query. With adaptivity, however, a peer is clustered together with many neighbors with the same interests. Thus it is unlikely for a query to be dropped before it has reached most peers that are able to answer it. Indeed, we see in Figure 8.4 that with adaptivity, RTR_{ef} achieves the same number of useful query messages as all-forward and RTR_{nf}. Furthermore, because RTR_{ef} is still able to intelligently route messages within these clusters via filters, it results in much fewer useless messages generated under all-forward.

Interest Overlap

Let us now consider the interest overlap between peers, in the case where there is *no adaptivity* (with adaptivity, competition is a problem regardless of interest overlap). Because peers always have an incentive to adapt under the RTR protocol, here we are considering a scenario in which adaptivity is not possible – e.g., an ad-hoc network. In our experiments, we set a number of T_{cat} total content categories ($T_{cat} = 20$ by default), and peers are "interested" in an average of N_{cat} categories, selected at the beginning of the simulation. We say that a peer is interested in a category if it contains files in that category. Figures 8.5 and 8.6 show us the overall performance of the network in terms of score and messaging overhead, respectively, while we vary the number of category interests per peer, N_{cat}, along

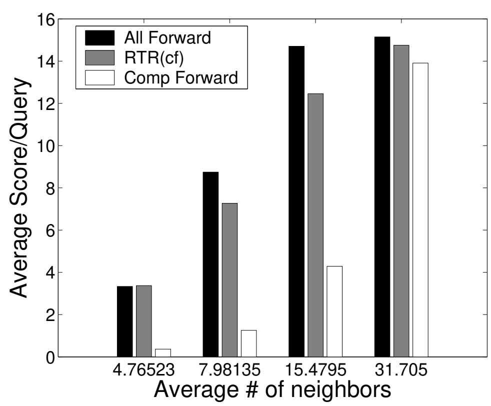

Figure 8.7: Competition becomes less of a problem with average neighbors increases

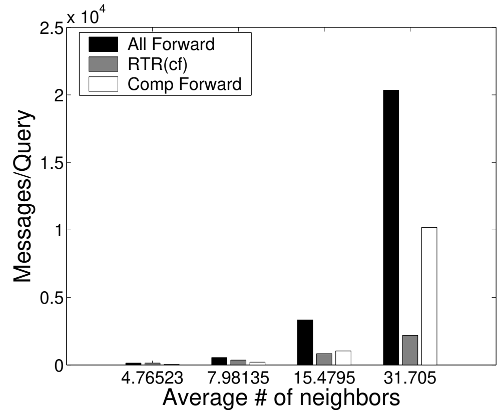

Figure 8.8: Messaging overhead is much smaller for RTR than for all-forward

the x-axis. We show the performance of both all and competitive-forward, as well as the performance of RTR_{nf}.

As the average number of category interests increases, so does the likelihood that a peer will have interests similar to other peers (i.e., greater interest overlap), and therefore be in competition. Indeed, we see from Figure 8.5 that as the number of category interests per peer increases, the difference between competitive and all-forward increases dramatically. Scores under all-forward increase because more peers have more answers to a given query, whereas scores under competitive-forward decrease, because query messages are not being forwarded. When $N_{cat} = 16$, the average query score under competitive-forward is just 5.8% of the score under all-forward.

Fortunately, in Figure 8.5, we also see that the RTR protocol does an excellent job in maintaining high quality of results. For all values of N_{cat}, the performance of the RTR protocol closely mirrors all-forward. In terms of cost, Figure 8.6 shows that while the RTR protocol is more expensive than all-forward, it is still within reason for the general case. For example, in the default case where $N_{cat} = 2$, the RTR protocol is about 43% more expensive than all-forward in terms of messages sent.

Number of connections

In the competitive-forward scenario, a peer does not receive queries because competitive neighbors drop them. Intuitively, then, the greater the number of neighbors each peer has, the less likely it is for a query to be dropped prematurely. In the extreme case, if the topology is strongly connected, competitive-forward will have the same performance as all-forward.

In Figures 8.7 and 8.8, we show the results of several simulation runs in which the average number of neighbors are varied across the runs. Because we could not control the number of neighbors directly, the average number of neighbors, shown along the x-axis, are not integers. For each number of average neighbors, these figures show us the performance of all-forward, competitive-forward and RTR_{ef}, measured in terms of average query score and messaging overhead, respectively.

Figure 8.7 confirms our intuition that competition becomes less of a problem as the number of connections increases. For example, when the average number of neighbors is roughly 8, competitive-forward has an average query score that is just 14% of all-forward and RTR_{ef}. However, when the average number of neighbors is 31, competitive-forward and all-forward have the same quality of results. In Figure 8.8, we see that competitive-forward also incurs less overhead than all-forward.

We do note, however, that as network size increases, more neighbors are required before competitive and all-forward have similar performance. When average neighbors is 16 in our simulations, all-forward is sending the query to every peer in the network; in a larger network, quality of results for all-forward will continue to improve with higher connectivity (> 16 connections). Therefore, it will also take higher connectivity for competitive-forward to "catch up" to all-forward. We also note that a network cannot always simply increase connectivity to overcome competition: it is generally not reasonable to allocate hundreds of open connections to a single application, and networks may face other real-world constraints on outdegree, such as proximity in an ad-hoc network.

Again, the RTR protocol does an excellent job at maintaining the same quality of results at a very low cost. For example, when average number of neighbors is 16, the RTR protocol results in almost the same quality of results as all-forward, but at just a fourth of the cost. Even when non-cooperation is no longer a problem (e.g., # neighbors is 31), the RTR protocol is still very useful in dramatically reducing messaging overhead by almost 90%. This surprising observation is again explained by the fact that the RTR protocol routes queries selectively, rather than generating many messages (and duplicates) by blind broadcast. In particular, almost the entire increase in cost for all-forward is accounted for by duplicate messages.

In conclusion, when peers adapt, interests overlap, or number of neighbors is limited,

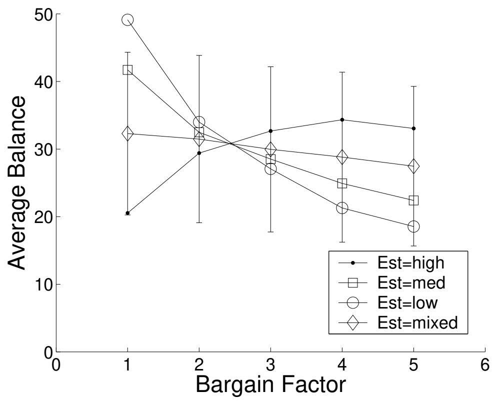

Figure 8.9: Individual balance for various values of $f_{bargain}$

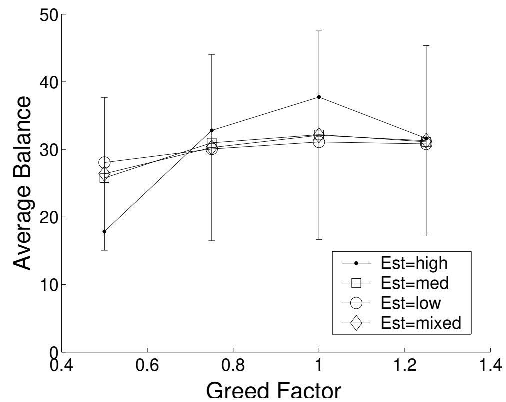

Figure 8.10: Individual balance for various values of f_{greed}

non-cooperation renders the network ineffective. We have shown the RTR protocol to effectively address non-cooperation in these cases, and to be very useful in all cases in reducing messaging overhead while maintaining excellent quality of results.

8.5.2 Parameter Selection

In Section 8.4 we presented a behavior model with 5 parameters that define an individual policy: f_{filter}, f_{greed}, $f_{bargain}$, p_{nodup} and f_{return}. In a system where peers are autonomous, peers may select their own policies. In this section, we study how peers will choose parameter values, and the effect these choices have on overall network behavior. We note that f_{return} and p_{nodup} are simple estimations of system-wide values, therefore a rational policy will always be to estimate these as well as possible. Therefore we focus on the remaining three parameters.

Individual Effect

A peer's policy is largely reflected through $f_{bargain}$ and f_{greed}, which control the price at which an RTR is bought and sold, respectively, relative to the expected value. Unlike f_{filter}, which can be determined at design time (Section 8.3.4), $f_{bargain}$ and f_{greed} may be freely chosen by each peer, depending on preferences stated by the user.

Figure 8.9 shows an analysis of the impact of $f_{bargain}$ on peer balance. Each curve represents an experiment where each peer chooses its own value for $f_{bargain}$ (out of a few choices, shown along the x-axis), which it keeps for the entire simulation. Remaining parameters are assigned default values from RTR_{cf}. At the end of simulation, we calculate average ending balance of peers grouped by $f_{bargain}$ value. Standard deviation (shown by vertical bars for only the `Est = mixed` curve, for clarity) is large because of the heterogeneous nature of peers – e.g., some peers have many more files than others. Because the impact of $f_{bargain}$ depends largely on the quality of estimated values of RTRs, we ran four experiments (one per curve) where peers have uniformly high, medium, low, or mixed estimates. Estimates are influenced by choosing values of f_{return} above, at, or below true value.

Figure 8.9 shows us that, in general, expected income is maximized by dealing fairly, rather than bargain hunting. The exception is when estimates are uniformly high. If a peer discovers that every one else's assessment is too high, the "correct" response is not to increase $f_{bargain}$ as Figure 8.9 might suggest, but rather to adjust f_{return} or p_{nodup} to reflect a more realistic model. In doing so, the peer still buys RTRs at a lower price (as if it just increased $f_{bargain}$). In addition, the peer will have a more accurate model with which to make informed decisions. Hence, as long as a rational peer believes his model to be reliable, he will choose $f_{bargain} = 1$ to maximize income.

Figure 8.10 shows a similar experiment as Figure 8.9, but where peers only vary in f_{greed}. Here we see that f_{greed} has relatively little impact on peer balance (again, unless estimates are uniformly high) because increasing f_{greed} presents a tradeoff between higher income per sold RTR, but fewer RTRs sold. For example, when estimations are uniformly high, peers with $f_{greed} = .5$ sold an average of 1100 RTRs per simulation, while peers with $f_{greed} = 1.25$ sold 630. This tradeoff does result in a slight maximum at $f_{greed} = 1$. Hence, rational peers are most likely to select $f_{greed} = 1$.

Discussion. For simplicity, the above analysis of f_{greed} and $f_{bargain}$ assumes that each peer uses a single parameter value for all decisions. However, in reality, a peer may use different values on a *per peer, per RTR* basis. For example, in a no-filter system, a peer B may charge neighbor A a higher price than A would likely pay if A did not own the file. If A buys the RTR, then B might conclude that A owns the file with some high probability. In return, a peer may also refuse to buy an RTR from a particular neighbor, even if the price is less than the expected value (i.e., act as a bargain-hunter), in order to fool its neighbor's "learning model" into charging lower prices in the future.

We leave as future work a formal analysis of this interaction between f_{greed} and $f_{bargain}$. For now, we just make the following high-level observation: a peer must be profitable to its neighbor, otherwise it will lose neighbors and be pushed to the edge of the network.

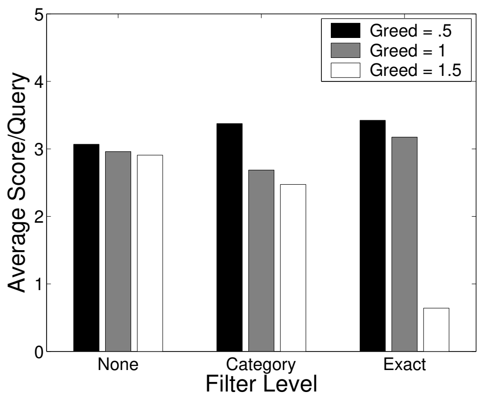

Figure 8.11: Average query scores for varying parameter values

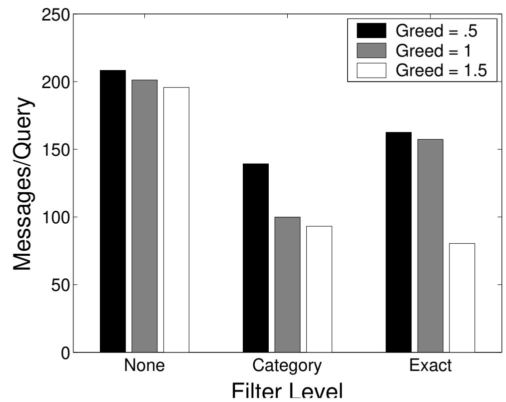

Figure 8.12: Messaging overhead for varying parameter values

To a certain degree a peer may "outsmart" its neighbors to maximize income. Over time, however, extreme behavior of either type – especially if the magnitude exceeds that of other peers in the system – will make the peer unprofitable to its neighbors.

Overall Impact

Let us now consider how parameters impact overall system performance. Ideally, we will find that the values chosen naturally by selfish peers will result in best overall performance as well.

First consider f_{greed} and f_{filter}. In Figure 8.11 we see the outcome of 9 different experiments, each running the RTR protocol with different values of f_{filter} and f_{greed}. For example, in the experiment where no filters are used and $f_{greed} = .5$, all peers used these same values for f_{filter} and f_{greed}, and default values for the remaining parameters.

With regards to f_{filter}, Figure 8.11 confirms what we found earlier in Section 8.5.1: filters allow queries to be intelligently routed to those peers with answers; therefore message cost is greatly reduced. We also find in Figure 8.11 that *price inflation* is harmful to system performance – i.e., when $f_{greed} > 1$. When exact filters are used, price inflation ($f_{greed} = 1.5$) results in just 19% the average query score achieved by $f_{greed} = .5$. Note, however, that price inflation only comes into play when filters are used. Category and exact filters are sensitive to f_{greed}, because with filters, prices more accurately estimate the true value of the RTR to the neighbor. Thus, price inflation is more likely to cause neighbors to drop offers (due to price exceeding value), than if no filters are present.

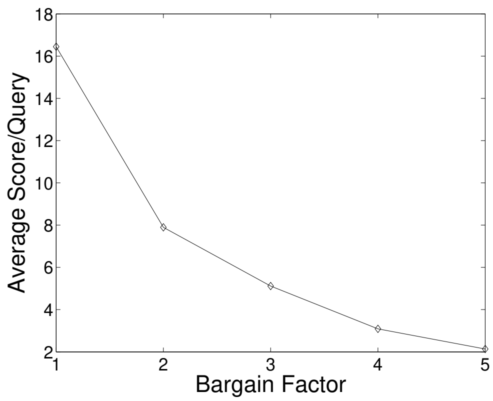

Figure 8.13: Average query scores for varying $f_{bargain}$ values

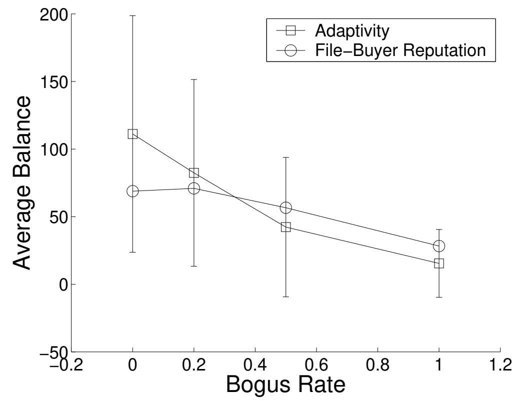

Figure 8.14: Individual peer balances using adaptivity

With regards to $f_{bargain}$, we found low values of $f_{bargain}$ to result in best overall performance. In Figure 8.13, we see the average query score in the network as $f_{bargain}$ is varied. Each point in the figure represents a network in which all peers have the same value of $f_{bargain}$. From this figure, we see clearly that $f_{bargain} = 1$ has best overall effect. Larger values of $f_{bargain}$ are harmful because they stifle the economy: they reduce the number of peers who buy RTRs, and hence, the number of peers able to respond to queries. Furthermore, lower value for $f_{bargain}$ result in a better quality-cost ratio. While $f_{bargain} = 1$ results in average query score that is 8 times higher than $f_{bargain} = 5$, average messaging overhead is only four times higher (experiment not shown here).

Fortunately, we found in Section 8.5.2 that $f_{greed} = 1$ and $f_{bargain} = 1$ result in highest individual utility. Therefore, we see that under the RTR protocol, peers will tend to choose behavior that results in desired overall system outcomes.

8.5.3 Handling Bogus RTRs

In Section 8.3.4, we discussed two techniques to combat bogus RTR "attacks" on the RTR protocol: adaptivity and file-buyer reputation. Here, we wish to analyze the effectiveness of these techniques. A technique is *effective* if peers can maximize income by minimizing the rate at which it issues bogus queries.

Figure 8.14 shows us the individual balances of peers that issue bogus queries at varying rates, when adaptivity only, and file-buyer reputation only (with *no* adaptivity), are used. For these experiments, peers issue bogus RTRs at different rates: 0%, 20%, 50% and 100%.

A $x\%$ bogus rate means $x\%$ of a peer's queries are bogus. We show the average balance of all peers with a given bogus rate at the end of our simulations. Again, standard deviation (denoted by vertical bars, shown for the adaptivity curve only, for clarity) is large because peers are heterogeneous – e.g., some peers have many more files to sell than others.

In Figure 8.14 we see that both adaptivity and file-buyer reputation are effective in rewarding good behavior and punishing bad behavior, with adaptivity more prominently rewarding good peers. For example, average balance of a good peer (with a bogus rate of 0%) is almost five times higher than the average balance of a completely cheating peer (with bogus rate of 100%) under adaptivity.

Adaptivity is effective because, as expected, it pushes misbehaving peers to the edge of the network. Although all peers start with the same number of neighbors on average, at the end of the simulation, peers with high bogus rates have much fewer neighbors than peers with low rates. In addition, peers at the edge of the network receive lower scores for the valid queries they issue, due to weaker connectivity to the rest of the network. For example, peers with 0% bogus rate had an average of 11 neighbors each, and average individual query score of 5. Peers with 100% bogus rate had an average of 2 neighbors, and average score of 1.1.

Similarly, file-buyer reputation is effective because reputation scores are accurate: in our simulations, a peer's reputation was roughly linear with its bogus rate. However, observe in Figure 8.14 that while peers with extremely high bogus rates are punished, those peers with moderate bogus rates (e.g., 20%) have nearly as good performance as good peers that never submit bogus queries. We believe such an effect is actually *good*. Even good users will likely submit bogus queries occasionally – e.g., if they do not find a satisfactory answer to their query. Therefore, we want the punishment for bogus RTRs to be "light" until bogus rate becomes excessive (e.g., $> 50\%$).

8.6 Conclusion

In summary, we have shown that the *non-cooperation problem* presents a significant challenge to competitive P2P networks. We presented one promising solution, the *RTR protocol*, that gives peers the incentive to cooperate in the operation of the network, even in the face of competition for providing services. We have shown how our protocol enjoys a higher quality-cost ratio than even the non-competitive scenario, by efficiently routing queries to peers that can provide good answers, and how the protocol is robust against cheating peers.

Chapter 9

Conclusions and Future Work

This thesis presented our work towards efficient and incentive-compatible peer-to-peer resource discovery.

Our first focus was on developing a realistic model of a P2P file-sharing application (Chapter 2), which then served as an evaluation tool for our later work. In developing this model and validating it over live, large-scale P2P file-sharing systems, we gained an understanding of typical system load, user behavior and expectations.

Next, we investigated efficient search mechanisms in a cooperative environment. In Chapter 3 we presented a number of *hybrid* search architectures and login policies, and evaluated them through extensive simulations. Not surprisingly, the architectures and policies had different strengths; through our findings, we were able to recommend the best architectures for the current and various projected application scenarios.

In pure networks, one of the most challenging aspects of search is *message routing*. We studied two message routing protocols in Chapter 4 – *Iterative Deepening* and *Directed BFS* – that improve efficiency by adapting to the dynamic characteristics of a peer's query and neighborhood. We also described *Local Indices*, a simple index-replication scheme that pushes a peer's local index to a small number of neighbors. We showed that all three techniques substantially reduce aggregate cost of query processing, while maintaining equally high quality of results.

Another important aspect of search is *role differentiation*: assigning different roles to peers according to their capacities, in order to decrease overall system load. In Chapter 5 we presented a global design algorithm for super-peer networks that recommends an efficient topology and assignment of roles, given individual peer constraints as input. We also describe several local decision-making policies peers can follow to produce a globally efficient role assignment.

In Chapter 6, we studied a new type of search architecture, in which messages are *not forwarded*; instead, peers keep a large cache of pointers to other peers, which they contact directly during a query. We refined the GUESS protocol, and identified key peer decisions that can greatly affect the performance of the system and the quality of results received by the peer. By defining several simple *policies* to govern these decisions, we were able to improve the efficiency of the system by several orders of magnitude.

In the second part of the thesis, we switched our focus to incentive-compatibility – designing systems that not only promote efficiency, but encourage participation and correctness as well. Chapter 7 described PPay, our micropayment protocol that exploits unique characteristics of P2P applications to greatly decrease broker load while maintaining good security properties. Our work in Chapter 7 can be used not only for incentives in a search mechanism, but can also form the basis for any economic P2P application in which small, lightweight transactions are common.

Finally, in Chapter 8, we described the RTR protocol, which promotes peer participation in a search mechanism by having peers sell the right to respond to queries. Our protocol not only overcomes non-cooperation by providing proper incentives to peers, but also results in a network that is efficient through intelligent, incentive-compatible routing of messages.

While this thesis takes several steps towards efficient, incentive-compatible resource discovery in peer-to-peer systems, there are many interesting subtopics that remain to be investigated. We conclude by describing some remaining open problems in the area.

Incentive-Compatibility

Our study of the RTR protocol (Chapter 8) provided an initial investigation of incentive-compatible message routing. In the future, we would like to gain a better understanding of individual peer choices through the application of game theory, and also to consider solutions for non-cooperation over alternative search architectures, such as GUESS and DHTs. We would also like to extend incentive-compatibility to other aspects of efficient resource discovery, such as caching, replication and indexing.

Expressive Queries with Guarantees

Our work on efficient query processing in Chapters 4 – 6 focused on simple "partial lookup" queries, where users are interested in receiving a minimum of k results, but not necessarily *all* results. While partial lookups are useful in a number of scenarios, and compensate partly for the difficulty of processing queries over an unreliable peer base, they may not be sufficient for certain applications. For example, an information-retrieval system often requires all results, or the top k (as opposed to first k). A distributed database system would require all results. An important direction for future research would be to investigate processing of complex queries over pure networks, with guarantees on what is returned.

Due to the "best effort" nature of unstructured systems (defined in Section 1.2), queries with guarantees will probably require borrowing techniques from existing work in structured P2P systems. Existing research in structured overlays has mostly focused on simple lookups (e.g., [85]), range queries (e.g., [5]), and multi-dimensional queries(e.g., [33]). There has also been work on complex query languages, such as SQL [45] and XML [31]); however a good case is yet to be made on the efficiency and feasibility of such P2P systems.

Flexible Score Management

Regarding PPay (Chapter 7), two interesting issues remain open. First, PPay currently handles a single denomination of coins; an important extension would be to allow efficient handling of multiple denominations. Second, risk management and reduction for both peers and brokers merit further investigation, particularly in the area of soft credit windows.

At a higher level, while PPay provides good security and is extremely efficient in comparison to existing micropayment schemes, we should theoretically be able to improve efficiency further if we are willing to give up certain security guarantees. Ideally, an application can decide what level of security it desires, and select the appropriate score management scheme that meets its needs at the lowest cost. One new and interesting area of research is developing a family of *score management* schemes that allow flexibility in the tradeoff between security and efficiency.

For example, consider a system in which users can trade different types of digital resources: compute cycles, storage space, information/data files, etc. Rather than directly trading resources, however, users have the flexibility to buy and sell resources using tokens, which may not be cashed in for real money. Clearly, the security guarantees regarding these tokens can be much lower than those regarding real money, as users will be less upset if they are not compensated for uploading a file than if they actually lose a dollar. Therefore, such a system would select a score management scheme that is more lightweight than PPay, and which provides "good enough" security for its needs.

Peer-to-peer systems present a new paradigm of distributed system design with enormous potential. Building upon the work presented in this thesis and the above avenues of future research, we hope these systems will come closer to fulfilling their promise of efficient, effective resource sharing.

Appendix A

The Overhead of Open Connections

As the number of open connections increases, the operating system overhead of handling messages increases as well. We model the cost of this overhead as follows.

First, we assume a node (e.g., a super-peer or hybrid server) is designed as an *event-driven* server, where a single thread services all connections using the `select` system call to detect events on these connections. Event-driven systems are often preferred over thread-per-connection designs because of simplicity of design, and lower context-switching and synchronization overhead, especially on single-processor machines. In many operating systems (e.g., Digital UNIX, 4.3BSD, Linux), the implementation of `select` does a linear scan over an array of file descriptors, where there is one descriptor per open connection. The cost of scanning a single file descriptor in this array has been reported to be 3μs on a Pentium 100 machine running Linux 2.1.x in [37], which translates to about .04 units in our scale.

Now, reference [6] shows that as the number of open connections increase, the cost of a `select` call also increases linearly until the number of connections exceeds roughly 1000, after which the cost levels off. This leveling off is due to the amortization of events across calls to `select`. That is, as the number of connections increases, the time spent in the `select` call increases, so that more events will be reported per call. However, the load of events in these experiments is much higher than the load experienced by a node in the default configuration specified in our experiments (the exception being hybrid servers, which have roughly constant overhead due to open connections). Hence, a node would probably continue to see a linear cost growth for much larger numbers of connections.

Furthermore, none of the results reported in our results consider the case where the number of open connections exceeds 1000, so for the range of connections we consider, the linear cost growth will be present.

A linear growth of overhead, however, does not imply that there will be a single call to `select` per message. Let us assume that in the case where the number of open connections is very large (approximately 1000) and TTL=1. In this case, roughly 1 out of 4 incoming/outgoing messages are incoming query messages, which take the longest to process. We will then assume that once a query has been processed, an average of 4 more events will be discovered by the next call to `select`, so that the cost of the call will be amortized over 4 messages. Hence, additional overhead per open connection per message is $\frac{.04}{4} = .01$ units.

Appendix B

Hybrid Systems: Additional Analysis and Results

B.1 Finding *ExServ* for the Hash Architecture

Let $P(b, i, k)$ be the probability that b inverted lists reside at exactly i servers when there are k total servers. Then, *ExServ* is equal to:

$$ExServ = \sum_{s=1}^{k} s \cdot P(\mathit{WordsPerQuery}, s, k) \tag{B.1}$$

Note that, unlike in the expression for *ExServ* in Section 2.3.2 for the chained architecture, it does not make sense here to include expressions for $s > k$.

The problem of finding $P(b, i, k)$ can be restated as the probability of b balls being assigned to exactly i bins, where each ball is equally likely to be assigned to any of the k total bins. When b is small, we calculate this value directly. If b is a fraction, then we interpolate the value. For example, if $b = \mathit{WordsPerQuery} = 2.4$, then we first calculate *ExServ* for $b = 2$, then *ExServ* for $b = 3$. Let $k = 5$. Then, when $b = 2$,

$$P(2,1,5) = \binom{5}{1}\left(\frac{1}{5}\right)^2 = .2 \qquad P(2,2,5) = \binom{5}{2} 2! \left(\frac{1}{5}\right)^2 = .8. \tag{B.2}$$

Then, $ExServ = 1 \cdot .2 + 2 \cdot .8 = 1.8$.

Similarly, when $b = 3$,

$$P(3,1,5) = \begin{pmatrix} 5 \\ 1 \end{pmatrix} \left(\frac{1}{5}\right)^3 = .04 \qquad P(3,3,5) = \begin{pmatrix} 5 \\ 3 \end{pmatrix} 3! \left(\frac{1}{5}\right)^3 = .2 = 48 \qquad \text{(B.3)}$$

and $P(3,2,5) = 1-(P(2,1,5)+P(3.3.5) = 0.48$. Then, $ExServ = 1{\cdot}.04+2{\cdot}.48+3{\cdot}.48 = 2.44$. Now, we interpolate a value for *ExServ* when $b = 2.4$:

$$ExServ = 1.8 + (2.44 - 1.8) \cdot (2.4 - 2) = 2.056. \qquad \text{(B.4)}$$

When b is not small, then the expression for $P(b,i,k)$ becomes complex. In this case, we use a conservatively large approximation for *ExServ*. We know that if there is only one ball ($b = 1$), then $ExServ = 1$. If there are two balls ($b = 2$), then *ExServ* increases by a value of $1-\frac{1}{k}$. As more balls are added, *ExServ* increases by an increasingly smaller value. However, we estimate *ExServ* by assuming that for every ball added, *ExServ* will continue to increase by $1 - \frac{1}{k}$. For example, in the hash architecture, every word in a file's metadata is hashed to a server, and that file must be replicated at that server. To find at how many servers a file is expected to be replicated, we compute the value $1 + (WordsPerFile - 1)(1 - \frac{1}{k}) = 1+(10-1)(1-\frac{1}{5}) = 8.2$. We then take the maximum of this value, and the actual number of servers. Hence we expect a given file to be replicated at $max(8.2, 5) = 5$ servers on average. The estimate for *ExServ* will be higher than the true value, but as long as k is small or b is not too large, the estimate will still be close.

B.2 Miscellaneous Experiments

In this section, we present additional experiments help us to better understand the strengths and weaknesses of each hybrid strategy presented in Chapter 3.

Performance Impact of MaxResults. Existing file-sharing systems in the music industry set a rather low limit on the number of results returned for a query. Setting a limit is good practice because queries are narrow, and limiting the number results can save bandwidth while not preventing the user from finding what she is looking for. For example, searching for "erasure always" on Napster will return 100 copies of that same file. However, file-sharing systems in other domains may not have the same kind of narrow search.

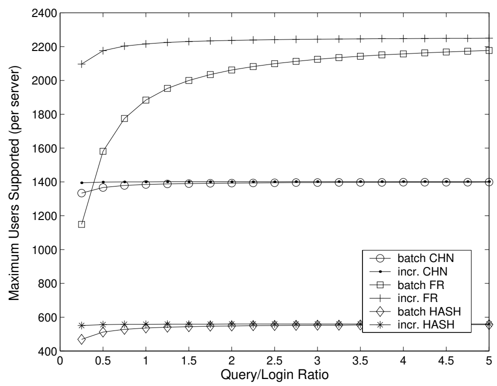

Figure B.1: Performance as *QueryJoinRatio* is varied, with MaxResults $= \infty$

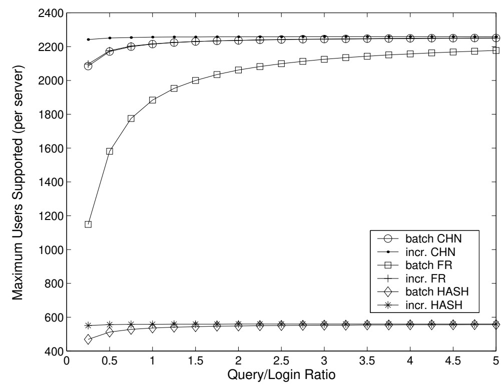

Figure B.2: Performance as *QueryJoinRatio* is varied, MaxResults $= \infty$ and Chained-Direct

Instead, searches are on a topic, and the user wants all results returned. For example, if the domain were photography, and a user wished to search for photographs of "yellow cars," every photograph returned would be unique, and the user might want to see every one before deciding which ones to update. In such a scenario, *MaxResults* is effectively set at ∞.

When *MaxResults* $= \infty$, inter-server bandwidth can become a problem. For CHN, remote results are sent from remote server to local server before being returned to the user. For HASH, inverted lists must be sent between servers. For FR, however, queries do not consume any inter-server bandwidth because every local server contains a complete index. Hence, we expect FR to perform the best in this scenario. Figure B.1 shows the performance of architectures when *MaxResults* $= \infty$, and inter-server communication is over WAN, along with user-server communication.

In this scenario, every architecture is bound by the combined network bandwidth. As expected, we see in the figure that FR performs the best, with the incremental FR strategy having over 55% better performance than both CHN strategies, and batch FR having nearly as good performance as incremental when *QueryJoinRatio* increases (because the expensive logins become less frequent). Again, HASH performs the worst in network bandwidth.

One might argue that CHN could save much bandwidth by having remote servers return results directly to the user, rather than routing them through the local server. The cost of such an implementation, which we will call "chained-direct," would be added complexity to the client. Figure B.2 shows performance of the strategies in the same scenario, but where the chained-direct implementation is used for CHN. We can see that chained-direct has only negligibly better performance than incremental FR, and again, batch FR has close to as good performance with larger *QueryJoinRatio*. The reasons chained-direct performance is not significantly better than FR are (1) queries must still be sent from server to server, even if results are not, and (2) with a large enough *QueryJoinRatio*, the extra inter-server communication used by FR's login messages is small in comparison to all the other network costs (e.g., user-server communication). In fact, FR can still perform better than chained-direct if *QueryJoinRatio* is quite large, and *FracChange* is small.

Another benefit of FR is query response time. Although we do not study this aspect of performance, FR would clearly have the best response time since all processing occurs at the local server only. All other architectures require relatively slow inter-server communication. Of course, FR also has the slowest login response time. However, this is generally considered to be a less important performance metric from the user's point of view, especially with higher *QueryJoinRatio*.

In summary, from these experiments we learn that:

- When remote query result sets are large, either in number of results or in size in bytes, then FR is recommended because it is least affected by query bandwidth usage.
- If, in the CHN architecture, we send remote results directly from a server to a remote user, then CHN and FR have comparable performance, but the CHN client contains additional complexity.
- Although not quantitatively considered in this chapter, query response time is an important performance metric, and FR has the best performance in this respect.

Other Sensitivity Analysis. In Section 2.2 we promised sensitivity analysis on *ActiveFrac* and *FracChange*, as these values could not be determined by the OpenNap experiments. The effects of the two parameters were summarized in the discussion of Figure 3.4. *ActiveFrac* and *FracChange* affect only the incremental strategy. *ActiveFrac* affects only queries, and *FracChange* affects only logins. As *ActiveFrac* increases, a larger fraction of the entire user population is online at any given time. Hence, fewer useless list accesses are made for every query, decreasing query costs. When *ActiveFrac* is low then queries are more expensive, so query-intensive architectures such as CHN and HASH perform best. As *FracChange* decreases, fewer offline changes are made to libraries, and hence, less work and bandwidth

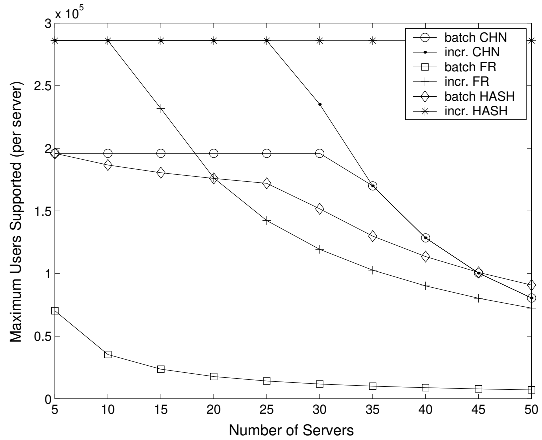

Figure B.3: Scalability with low *WordsPerQuery*

is needed for logins. When *FracChange* is low then logins are expensive, so login-intensive architectures such as FR perform best.

Single-Word Query Systems. In all scenarios considered so far where inter-server bandwidth is limited, HASH has very poor performance because of the large amount of list data transferred during queries. The amount of data transferred is a function of how many terms are in the query; in the worst case, *WordsPerQuery* - 1 inverted lists are transferred over the inter-server connection per query. In some systems, *WordsPerQuery* is very low. For example, systems that support only search by key or ID will have *WordsPerQuery* = 1. In these systems, *WordsPerQuery* - 1 is zero or very small, in which case HASH can have good performance because little data is transferred between servers.

Figure B.3 shows overall performance in the same scenario as Figure 3.7, but with *WordsPerQuery* = 1.1. In this figure, we see that HASH is finally free of the inter-server bandwidth bottleneck for queries, and therefore has the best performance and scalability of all the strategies, particularly incremental HASH. Batch HASH is still bound by poor inter-server communication performance for logins, however, its performance is fairly scalable compared to the other strategies. In contrast, CHN starts off bound by user-server communication, but soon becomes bound by inter-server communication because of all the remote queries and remote results sent between servers. HASH does not suffer from this problem because, as discussed in Section 3.3, *ExServ* is fixed and *ExRemoteResults* is zero for HASH. In addition, FR is largely CPU bound, since login performance degrades rapidly

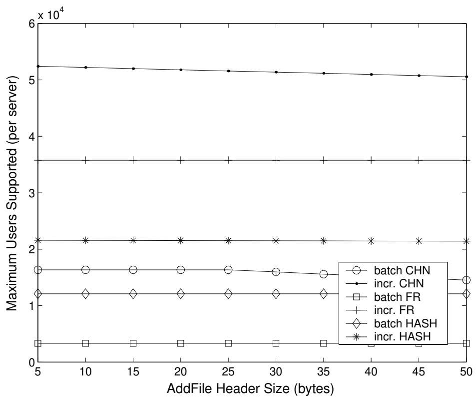

Figure B.4: Performance as AddFile packet header size is varied

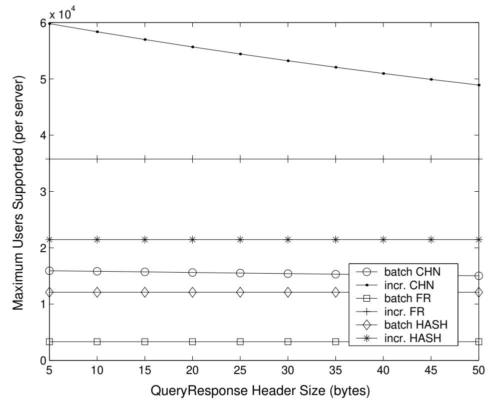

Figure B.5: Performance as QueryResponse packet header size is varied

as *NumServers* increases. As a result, HASH is clearly the recommended strategy when *WordsPerQuery* is close to 1.

Network Headers. In Section 3.4, we calculated the network bandwidth consumed by the system without considering network headers. The main reason for doing so was because we felt it was inefficient to send each AddFile message as a separate packet during login, and each QueryResponse message as a separate packet during query. Rather than redefine the network protocol, we decided to focus on the payload data, which varied between architecture and strategy, and defer the issue of headers until later. Now, we will discuss the impact of packet header size on system performance and show that while header size affects the absolute performance of the strategies, they do not affect their relative performance, which was the basis of our analysis on their strengths and weaknesses.

To simulate the effects of changing the network protocol, we can vary the average header size for the various message types. For example, a typical packet header is approximately 40 bytes for TCP/IP and Ethernet wrappers, and say we want to study the performance of the system if the protocol were defined such that an average of 10 AddFile messages were sent in a single packet. That is, instead of sending 168 AddFile packets to the server on login, the average client would only send 16-17 packets instead. To simulate the case where there are 10 messages per packet, we would evaluate system performance where average header size for an AddFile message is $40\frac{bytes}{packet}/10\frac{messages}{packet} = 4\frac{bytes}{message}$.

Figures B.4 and B.5 show the overall performance of each strategy when the header size for AddFile and QueryResponse is varied. We only analyze these two message types because for the music-sharing scenario, they account for over 98% of all user-server packets with the batch login policy, and over 95% with the incremental policy. All parameters are the defaults listed in Tables 3.1 and 3.2. In these figures, all packet header sizes other than the one being varied are assumed to be 40 bytes. As seen in these figures, packet header size does affect actual performance of some strategies, but does not affect the relative ordering of the strategies. Furthermore, the actual performance of most strategies is affected very little. Incremental and batch FR, which are bound by CPU, are not affected at all. The strategies that are most affected, batch and incremental CHN, are bound by user-server communication. User-server communication consists largely of many small packets, so a change in packet header size has a relatively significant impact on overall performance. However, note that HASH, which is bound by inter-server communication, is barely affected by a change in packet header size. This is because the bulk of the bandwidth used by HASH is consumed by payload data. Hence, packet header size has little or no affect on its overall performance.

In summary, we believe that network protocol and packet header issues are important to consider, to ensure the most efficient execution of the application. However, because our analysis in this section has been based on relative performance of the strategies and on characteristics of the performance graphs, we believe our analysis is unaffected by these issues.

Appendix C

Message Routing: Load Calculations

C.1 Aggregate Bandwidth Calculations

Iterative Deepening. To calculate the aggregate bandwidth consumed under the Iterative Deepening technique, we must take into account the cost of sending Resend messages, as well as the possibility that a query may terminate before reaching the maximum depth D.

First, let P be the Iterative Deepening policy we are evaluating. Recall that policies are defined as an ordered list of values, where the ith value tells us the depth at which the ith iteration ends, or freezes. We say a particular depth d is a "policy level" if $d \in P$.

Let us define following indicator functions:

$$I_1(Q,n,W,P) = \begin{cases} 1 & \text{if } n \text{ is a policy level, and } Q \text{ does } \textit{not} \text{ end at depth } n \\ 0 & \text{otherwise} \end{cases} \quad I_2(Q,n,W,P) = \begin{cases} 1 & \text{if } Q \text{ does } \textit{not} \text{ end } \textit{before} \text{ level } n \\ 0 & \text{otherwise} \end{cases} \quad I_3(n) = \begin{cases} 1 & \text{if } n < D \\ 0 & \text{otherwise} \end{cases} \tag{C.1}$$

For I_1, a query Q is said to "end" at depth n if n is a policy level, and the query source determines that Q has been satisfied after the query has reached a depth of n, but before the time limit has expired. Recall that the time allowed between iterations is W time units. Hence, if n is the ith value in policy P, then query Q "ends" at depth n if $T(Q, Z, W, P) <$

$i \cdot W$. Function $I_2(Q,n,W,P) = 0$ if and only if $\exists k$ such that $h(Q,k,W,P) = 1$, for $k < n$ – that is, iff query Q has ended at some previous policy level. Depth n does not need to be a policy level. Function I_3 is self-explanatory.

Now, the formula for calculating aggregate bandwidth of query Q with Iterative Deepening is as follows:

$$\begin{aligned} BW_{id}(Q,W,P) = & \\ & \sum_{n=1}^{D} I_2(Q,n,W,P) \cdot \Big(a(Q) \cdot \big(N(Q,n) + C(Q,n)\big) + n \cdot \big(c \cdot R(Q,n) + d \cdot M(Q,n)\big)\Big) \\ & + \sum_{x \in P} \Big(I_1(Q,x,W,P) \cdot I_2(Q,x,W,P) \cdot I_3(x) \cdot \sum_{n=1}^{x} b \cdot \big(N(Q,n) + C(Q,n)\big)\Big) \end{aligned} \tag{C.2}$$

The first summation gives us the bandwidth consumed by sending the query to level n, and receiving response messages from level n; it is almost identical to the summation in Equation 4.1. If the query ends before reaching level n, however, then it will not be sent to level n nor receive results from level n. So in the above equation, we multiply the terms in the first summation by the indicator function $I_2(Q,n,W,P)$.

The second summation gives us the bandwidth consumed by sending out Resend messages. Resend messages are sent to depth n given three conditions: (1) x is a policy level, (2) query Q is not satisfied at or before depth x within the time limit, and (3) $x < D$, since the Query message cannot reach a depth greater than D. The first condition is satisfied because we only sum over those levels x such that $x \in P$. The second condition is satisfied by multiplying the terms in the summation by $I_1(Q,x,W,P) \cdot I_2(Q,x,W,P)$, which equals 0 iff the query is satisfied at or before depth x. Finally, the third condition is satisfied by multiplying the terms by $I_3(x)$, which equals 0 if $n = D$.

Directed BFS. Calculating bandwidth for Directed BFS is almost identical to calculating bandwidth for regular BFS, except that now before each query, we must first determine which neighbor would have been selected by the heuristic we are evaluating. Let us define the function $Y(Q,h)$ to return the neighbor that a query Q is submitted to, under heuristic h. For each query Q and heuristic h that we consider, we determine $Y(Q,h)$ from our logs by looking at the statistics on each neighbor that were gathered before the query was submitted. For example, say right before query Q is submitted, we find that neighbor A has a queue length of 10, neighbor B has a queue length of 5, and neighbor C has a queue length of 11. If heuristic h is to choose the neighbor with the shortest message queue, we define $Y(Q,h)$ to return neighbor B.

We can now calculate the aggregate bandwidth of this query, under a heuristic h, simply by using Equation 4.1, but replacing the functions $M(Q,n)$, $R(Q,n)$, and $N(Q,n)$, with the analogous functions $M(Q,n,y)$, $R(Q,n,y)$, and $N(Q,n,y)$:

$$BW_{dbfs}(Q,h) = \sum_{n=1}^{D}\Big(a(Q)\cdot\Big(N(Q,n,y)+C(Q,n,y)\Big)+n\cdot\Big(c\cdot R(Q,n,y)+d\cdot M(Q,n,y)\Big)\Big) \tag{C.3}$$

where $y = Y(Q,s)$.

Local Indices. To calculate aggregate bandwidth consumed, let us define o_j =`QueryJoinRatio`, o_u =`QueryUpdateRatio`, and $m = MAX(p)$ for all $p \in P$. Bandwidth consumed is now calculated by the following formula, for a given radius r and policy P:

$$\begin{aligned} BW_{li}(Q,P,r) = \\ &\sum_{n=1}^{m} a(Q)\cdot\Big(N(Q,n)+C(Q,n)\Big)+\sum_{x\in P} x\cdot e(Q,r)\cdot N(Q,x) \\ &+\frac{1}{o_j}\Big(f\cdot\sum_{n=1}^{r}\Big(N(Q,r)+C(Q,r)\Big)+f\cdot\sum_{n=1}^{r}N(Q,r)\Big) \\ &+\frac{1}{o_u}\Big(g\cdot\sum_{n=1}^{r}\Big(N(Q,r)+C(Q,r)\Big)\Big) \end{aligned} \tag{C.4}$$

The first summation is the cost of sending the query to all nodes within a radius of m. The second summation accounts for the cost of transferring response messages back to the query source. Only nodes at policy levels execute the query, hence the summation is over policy levels only. At a policy level x, every node returns one response message of average size $e(Q,r)$, and this response message must be forwarded through x hops.

The third term accounts for the cost of joining and leaving the network. First, Join messages of size f are broadcast to all nodes within r hops. Next, when a node joins the network and sends out its Join message, all nodes that receive this message will in turn send their Join messages directly back to the joining node. The cost of joins and leaves are then amortized over the cost of a query by dividing the join/leave cost by o_j, the value of `QueryJoinRatio`.

Symbol	Cost (Units)	Description
s	1	Cost of transferring a Resend message
$t(Q)$	$1 + .007 \cdot L(Q)$	Cost of transferring a Query message
u	.5	Additional cost of sending a Response message when a result record is appended to the message
v	1.2	Base cost of transferring a Response message with no result records
w	1.1	Additional cost of processing a query per result discovered
x	14	Base overhead of processing a query
$y(Q,r)$	$.47 + .72\cdot$ # addr $.5\cdot$ # results	Cost of transferring a Response message under Local Indices
$z(Q,r)$	$14 + 1.1\cdot$ # results	Cost of processing a query under Local Indices
h	$14 + 10.7 \cdot$ # files	Cost of processing a Join message
j	160	Cost of transferring a Join message
k	$14 + 10.7 \cdot$ # files	Cost of processing a timeout (removing metadata)
l	1.4	Cost of transferring an Update message
Δ	30	Cost of processing an Update message

Table C.1: Costs of actions

The fourth term accounts for the cost of transferring update messages. The cost of an update is amortized over the cost of a query by dividing the update cost by o_u, the value of `QueryUpdateRatio`.

C.2 Aggregate Processing Power Calculations

Iterative Deepening. The formula for aggregate processing power is very similar to Equation C.2, the formula for calculating aggregate bandwidth in Iterative Deepening, since for every message transferred, there is usually a corresponding cost for transferring and processing the message:

$$\begin{aligned}
&PP_{id}(Q,P) = \\
&\quad \sum_{n=1}^{L} I_2(Q,n) \cdot \Big(t(Q) \cdot \big(N(Q,n) + C(Q,n) \big) + n \cdot \big(u \cdot R(Q,n) + v \cdot M(Q,n) \big) + \\
&\qquad \big(w \cdot R(Q,n) + x \cdot N(Q,n) \big) \Big) \\
&\quad + \sum_{x \in P} \Big(I_1(Q,x) \cdot I_2(Q,x) \cdot I_3(x) \cdot \sum_{n=1}^{x} s \cdot \big(N(Q,n) + C(Q,n) \big) \Big) \qquad \text{(C.5)}
\end{aligned}$$

The first summation gives us the cost of broadcasting the query and receiving results, corresponding directly to the first summation in Equation C.2. The first term of this summation gives us the cost of transferring the query message to level n. The second term gives us the cost of transferring the response messages back to the query source. For each Response message (there are $M(Q,n)$ Response messages at depth n), there is the base cost of transferring the message. In addition, for each result record (of which there are $R(Q,n)$), there is the additional cost of transferring those bytes. Finally, the third term in the summation gives us the cost of processing the query and generating the results.

The second summation gives us the cost of broadcasting Resend messages.

Directed BFS. Again, the formula for calculating processing power for Directed BFS is almost identical to the formula for calculating bandwidth for Directed BFS deepening given in Equation 4.1:

$$
\begin{aligned}
&PP_{dbfs}(Q,h) = \\
&\quad \sum_{n=1}^{D} \Big(t(Q) \cdot \big(N(Q,n,y) + C(Q,n,y) \big) + n \cdot \big(u \cdot R(Q,n,y) + v \cdot M(Q,n,y) \big) + \\
&\quad \big(w \cdot R(Q,n,y) + x \cdot M(Q,n,y) \big) \Big) \qquad \text{(C.6)}
\end{aligned}
$$

where $y = Y(Q,h)$.

The first term in the summation gives us the cost of sending the query to level n, and the second term gives us the cost of transferring the responses from level n back to the query source. The third term gives us the cost of generating the responses at level n.

Local Indices. The procedure used to calculate aggregate processing power for a query is similar to the procedure used to calculate aggregate bandwidth. First, the cost of a query is calculated, then the cost of a join and leave (i.e., timeout). The cost of the join and leave is then amortized over the cost of the query, using the parameter `QueryJoinRatio`. Again, let us define m to be the largest level specified in policy P. The formula for aggregate processing power for a query Q is:

$$
\begin{aligned}
&PP_{li}(Q,P,r) = \\
&\quad \sum_{n=1}^{m} t(Q) \cdot \big(N(Q,n) + C(Q,n) \big) + \sum_{x \in P} \big(x \cdot y(Q,r) \cdot N(Q,x) + z(Q,r) \cdot N(Q,x) \big) \\
&\quad + \frac{1}{o_j} \Big((h+k) \cdot \sum_{n=1}^{r} N(Q,r) + j \cdot \sum_{n=1}^{r} \big(N(Q,r) + C(Q,r) \big) + j \cdot \sum_{n=1}^{r} N(Q,r) \Big)
\end{aligned}
$$

$$+\frac{1}{o_u}\left(\Delta \cdot \sum_{n=1}^{r} N(Q,r) + l \cdot \sum_{n=1}^{r} \Big(N(Q,r) + C(Q,r)\Big)\right) \tag{C.7}$$

The first two summations are exactly analogous to the first two summations in the bandwidth calculation of Equation C.4. The first summation gives us the cost of transferring the query to all nodes within a radius of m, and the second summation gives us the cost of generating and transferring responses from the nodes at the policy levels.

The third term in the formula accounts for the cost of joining and leaving the network. First, when a node joins, all nodes within a radius of r will process the Join, which costs x units. Similarly, when the node leaves, all nodes within a radius of r will process the timeout, which costs z units. These costs account for the term $(h+k)\cdot\sum_{n=1}^{r} N(Q,r)$.

Next, these Join messages just mentioned are broadcast with a TTL of r, meaning they will be sent to every node within a radius of r, as well as every cycle-completing edge within a radius of r. Total cost of transferring these messages is therefore $j\cdot\sum_{n=1}^{r}\Big(N(Q,r)+C(Q,r)\Big)$.

Finally, when a node receives a Join message, it will generate a Join message over its own collection, and send this message directly to the joining node. The cost of transferring these messages is $j\cdot\sum_{n=1}^{r} N(Q,r)$. Note that we do not include the cost of processing these Join messages at the joining node, because we do not consider a node as part of the network until after it has processed all of these join messages. Hence, the cost of processing these joins is incurred on a node that is not yet part of the network, which we do not want to count.

The total cost of processing and transferring Join messages and timeouts is finally divided by o_j, the *QueryJoinRatio*, in order to amortize the cost of join/leave over the cost of a query.

The fourth term in the formula accounts for the cost of transferring Update messages and then processing these messages. This cost is then divided by o_u to amortize it over the cost of a query.

Appendix D

PPay: Extension Security Proofs

D.1 Security of Layers

Because layers over the coin are now used to encode reassignment, an owner U may no longer have the full audit trail of a coin. Invariant 3 as originally stated no longer holds. For example, say node V holds coin C owned by node U. V may reassign the coin to node X using layers, and then reassign the coin again to node Y using layers. U cannot detect V's misbehavior, because U is no longer involved in the reassignment.

However, we can show that the weaker form of invariant 3 still holds. Let us assume that node V is the nth holder of coin C owned by U, that V is the current valid holder, and that the three invariants hold. Say V now reassigns the coin to X via the layered coin P_{UVX}. Invariant 1 holds by the same argument used for the default protocol. Invariant 2 may not hold if V uses a bad sequence number, but X can detect this case and report misconduct to the broker. The broker will punish V and require it to produce a valid layered coin; hence V has no incentive to attack in this manner. Invariant 3 holds because an omniscient party now has access to the proofs stored as layers over a coin, as well as the proofs stored at the broker and the owner. We are given that the omniscient party can refute the first $n-1$ holders of C, and P_{UVX} now contains the proof of relinquishment by the nth holder. Therefore, the omniscient party can refute all old assignments of coin C.

Owner U becomes omniscient when the layered assignments of C are peeled at U (only one assignment of a coin should exist, but in the case of misconduct, then multiple assignments might exist). Unlike with downtime, owner U has no control over when it becomes omniscient. However, given that all coins must eventually be peeled to be cashed, such a time is guaranteed to exist. Furthermore, we can "hasten" this time by imposing a limit on the number of layers a coin may hold. Alternatively, coin renewals produce the same effect.

Finally, note that it is possible for the holder of a coin to "illegally" peel off layers of the coin, and proofs may be lost in this manner, thereby violating invariant 3. However, it is simple to detect and prove this type of misbehavior. Let us say V is the current valid holder of coin C, and that the assignment to V was made via a layered assignment P_{UTV} from node T. Note that the first field in an assigned coin (whether or not it is layered) is the recipient of the assignment. If V peels a layer off P_{UTV}, V will end up with a coin that is assigned to T. V can not change the recipient of the coin because we assume V can not forge the signature of whoever assigned the coin to T. Therefore, V can not use the peeled coin. If for some reason, V does peel the coin and tries to use it or cash it, the owner can prove that V misbehaved, and V will be punished. Furthermore, although invariant 3 no longer holds, V will be held financially responsible for any cheating that can be detected but for which no culprit can be identified (i.e., because the proofs were lost by V's peeling).

There is also the possibility that V did previously hold the layered coin, and V can peel off layers until it produces a layered coin assigned to itself. For example, consider the scenario in which the assignment A_{UV} is reassigned via layers as follows: $V \rightarrow X \rightarrow Y \rightarrow V$. The second time V receives the coin, it can peel three layers and reproduce A_{UV}. However, V has gained nothing by doing so – it could have easily reproduced A_{UV} by replicating that assignment. We have already discussed in Section 7.3.1 how replicating A_{UV} is not helpful, because it can not be reused. Therefore, U has no incentive to attack in this manner.

D.2 Security of Coin Renewals

After a renewal request, the holder V must check that the renewed coin has a renewal period in the future. Furthermore, ideally the renewal period adheres to a system-wide constant set by the broker. Because renewals present a tradeoff between performance, state and security (discussed below and in Section 7.6.2), renewal windows that are unnecessarily small will result in high loads for peer and broker, while renewal windows that are unnecessarily large will present security risks – especially if it is a malicious node who is trying to assign a long renewal period! Hence, in PPay we require that the renewal period be fixed for all coins. V must check that the new renewal period is correct after a renewal, and that the renewal period has not changed after a reassignment. If owner U tampers with the renewal period,

V can provide the original and reassigned/renewed coin to broker B as proof of misconduct on U's part.

To prove Theorem 1, we change the definition of a valid assignment to now include the restriction that the assignment has not expired. Furthermore, "proof of relinquishment" of an assignment is extended to include a renewal request for that assignment. Under this definition, we must show that all three invariants still hold under the coin renewal protocol.

Earlier, we showed how if the invariants held before the $n + 1$th reassignment, they will hold after the reassignment as well. Here, we show how if the invariants hold before a renewal request, they will hold afterwards as well.

Say node V is the valid holder of a coin C owned by U, and all three invariants hold. Now say V requests to renew its assignment A_{UV} via the renewal request:

$$N = \{A_{UV}\}_{SK_V} \tag{D.1}$$

At this point V is still the valid holder, but U possesses proof of relinquishment (i.e., the renewal request). This problem is analogous to the case in which V is having the coin reassigned, and submits a reassignment request. We deal with the problem in the same way, by having V go to the broker for renewal. Since U will be charged and cannot subsequently use the coin, it has no incentive to attack in this way. Eventually, invariant 1 will hold.

Invariant 2 holds because A_{UV} is still a valid assignment, and the serial number has not changed. Likewise, invariant 3 holds because the valid holder has not changed.

Now, owner U responds with the renewed coin:

$$A_{UV} = \{V, seq_n, ts, r'_b, r'_e, C\}_{SK_U} \tag{D.2}$$

Invariant 1 holds because owner U does not have proof that A_{UV} relinquished the renewed coin (distinguished from the old coin by the new renewal periods). Invariant 2 may not hold if U has changed the sequence number; however, just as with the new renewal period, V is responsible for checking the correctness of the sequence number and going to the broker if the value is incorrect. Eventually, then, invariant 2 will hold. Invariant 3 holds because the valid holder of the coin has not changed as a result of the renewal.

To show the correctness of the renewal policy, we also need to show that any proofs of relinquishment for expired assignments are not required for the three invariants to hold. Because destroying proofs can not create evidence against a valid holder, invariant 1 holds.

Furthermore, destroying proofs does not change the serial number of the current assignment, so invariant 2 holds. For invariant 3, any assignment that has expired is invalid, and the proof of the invalidity lies within the coin itself (the renewal end time, r_e). Hence, an omniscient party can automatically refute any assignment that has already expired. For any assignment that has not expired, but that has been relinquished (either reassigned or renewed), the omniscient party has not destroyed the proof of relinquishment. Hence, invariant 3 holds as well.

Bibliography

[1] L. Adamic, R. Lukose, A. Puniyani, and B. Huberman. Search in power-law networks. Available at http://www.parc.xerox.com/istl/groups/iea/papers/plsearch/, 2001.

[2] Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. http://www.firstmonday.dk/issues/issue5_10/adar/index.html, September 2000.

[3] H. Antwerpen. Electronic cash. Technical report, University of Eindhoven, Masters Thesis, 1990.

[4] N. Asokan, Matthias Schunter, and Michael Waidner. Optimistic protocols for multi-party fair exchange. Technical Report RZ 2892 (# 90840), 1996.

[5] James Aspnes and Gauri Shah. Skip graphs. In *Proceedings of the Fourteenth ACM-SIAM Symposium on Discrete Algorithms*, January 2003.

[6] G. Banga and J. Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proceedings of USENIX*, June 1998.

[7] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of Twentieth International Conference on Very Large Databases*, pages 192–202, September 1994.

[8] L. Buttyan and J. Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. In *ACM/Kluwer Mobile Networks and Applications*, 2003.

[9] D. Chaum and T. Pederson. Transferred cash grows in size. In *Advances in Cryptology – EUROCRYPT '92*, 1993.

[10] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the ACM SIGCOMM International Conference on Data Communication*, August 2003.

[11] E. Cohen, A. Fiat, and H. Kaplan. A case for associative peer to peer overlays. In *Proceedings of the First Workshop on Hot Topics in Networks*, October 2002.

[12] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the ACM SIGCOMM International Conference on Data Communication*, August 2002.

[13] T. Condie, S. Kamvar, and H. Garcia-Molina. Adaptive p2p topologies. Technical report, Stanford University, January 2004.

[14] B. Cooper, M. Bawa, N. Daswani, and H. Garcia-Molina. Protecting the pipe from malicious peers. Technical report, Stanford University, 2002. Available at http://dbpubs.stanford.edu/pub/2002-3.

[15] B. Cooper and H. Garcia-Molina. Bidding for storage space in a peer-to-peer data preservation system. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems*, July 2002.

[16] B. Cooper and H. Garcia-Molina. Ad hoc, self-supervising peer-to-peer search networks. Technical report, Stanford University, February 2003.

[17] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of Twenty-Second International Conference on Distributed Computing Systems*, July 2002.

[18] A. Crespo and H. Garcia-Molina. Semantic overlay networks for p2p systems. Technical report, Stanford University, May 2003.

[19] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, October 2001.

[20] E. Damiani, D. Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proceedings of the ACM Conference on Computer and Communication Security*, 2002.

[21] N. Daswani and H. Garcia-Molina. Query-flood dos attacks in gnutella. In *Proceedings of the ACM Conference on Computer and Communications Security*, November 2002.

[22] N. Daswani and H. Garcia-Molina. Pong-cache poisoning in guess. Technical report, Stanford University, August 2003.

[23] DigiCash website. http://digicash.com.

[24] P. Dutta. *Strategies and Games*. The MIT Press, 1999.

[25] Sandra Dykes. *Cooperative Web Caching: A Viability Study and Design Analysis*. PhD thesis, University of Texas at San Antonio, 2000.

[26] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A bgp-based mechanism for lowest-cost routing. In *Proceedings of the 21st Symposium on Principles of Distributed Computing*, July 2002.

[27] J. Feigenbaum, C. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63, 21-41 2001.

[28] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proceedings of the Sixth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, September 2002.

[29] D. Figueiredo, J. Shapiro, and D. Towsley. Using payments to promote cooperation in anonymity protocols. Technical report, University of Massachusetts, Amherst, 2003.

[30] Freenet website. http://freenet.sourcefourge.net.

[31] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *Proceedings of the Twenty-Ninth International Conference on Very Large Databases*, September 2003.

[32] P. Ganesan, Q. Sun, and H. Garcia-Molina. Apocrypha: Making p2p overlays network-aware. Technical report, Stanford University, November 2003.

[33] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in p2p systems. In *Proceedings of the Seventh International Workshop on the Web and Databases*, June 2004.

[34] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. The millicent protocol for inexpensive electronic commerce. In *Proceedings of the Fourth International World Wide Web Conference*, 1995.

[35] Gnutella website. http://gnutella.com.

[36] P. Golle, K. Leyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proceedings of the ACM Conference on Electronic Commerce*, October 2001.

[37] R. Gooch. I/o event handling under linux. http://www.atnf.csiro.au/ rgooch/linux/-docs/io-events.html, 2001.

[38] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kauffman Publishers, Inc., San Mateo, 1993.

[39] Grid computing info centre homepage. http://www.gridcomputing.com.

[40] GUESS protocol specification. http://groups.yahoo.com/group/the_gdf/files/-Proposals/GUESS/guess_o1.txt.

[41] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, October 2003.

[42] A. Gupta, B. Liskov, and R. Rodruigues. One-hop lookups for peer-to-peer overlays. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems*, May 2003.

[43] B. Horne, B. Pinkas, and T. Sander. Escrow services and incentives in peer-to-peer networks. In *Proceedings of Third ACM Conference on Electronic Commerce*, 2001.

[44] A. Huang, B. Ling, S. Ponnekanti, and A. Fox. Pervasive computing: What is it good for? In *Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access*, August 1999.

[45] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *Proceedings of the Twenty-Ninth International Conference on Very Large Databases*, September 2003.

[46] Apple - iTunes - Music Store website. http://apple.com/itunes/store.

[47] S. Jiang, L. Guo, and X. Zhang. Lightflood: an efficient flooding scheme for file search in unstructured peer-to-peer systems. In *Proceedings of the International Conference on Parallel Processing*, October 2003.

[48] S. Kamvar, M. Schlosser, and H. Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *Proceedings of the Twelfth International World Wide Web Conference*, May 2003.

[49] Brian Kantor and Phil Lapsley. Network News Transfer Protocol. RFC 977, February 1986.

[50] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the Eighth International World Wide Web Conference*, December 1999.

[51] KaZaA website. http://www.kazaa.com.

[52] Konspire Home Page. http://konspire.sourceforge.com/.

[53] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Thea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, November 2000.

[54] J. Kurose, M. Schwartz, and Y. Yemini. A microeconomic approach to decentralized optimization of channel access policies in multiaccess networks. In *Proceedings of Fifth International Conference on Distributed Computing Systems*, May 1985.

[55] J. Kytojoki and V. Karpijoki. Micropayments - Requirements and Solutions. http://www.tml.hut.fi/Opinnot/Tik-110.501/1999/papers/micropayments/-micropayments.html.

[56] Limewire website. http://www.limewire.com.

[57] S. McIlraith, T.C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems. Special Issue on the Semantic Web*, 16(2):46–53, March-April 2001.

[58] S. Micali and R. Rivest. Micropayments revisited. In *"CT-RSA"*, 2002.

[59] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[60] MojoNation website. http://www.mojonation.net.

[61] Morpheus website. http://www.musiccity.com.

[62] Napster website. http://www.napster.com.

[63] NetBill website. http://www.ini.cmu.edu/netbill.

[64] NeuroGrid website. http://www.neurogrid.net.

[65] C-Net News. Napster among fastest-growing net technologies. http://news.com.com/2100-1023-246648.html, October 2000.

[66] C. Ng, K. Sia, C. Chan, and I. King. Peer clustering and firework query model in the p2p network (poster). In *Proceedings of the Eleventh International World Wide Web Conference*, May 2002.

[67] Anh NgocVo and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *Proceedings of the Twenty-First International Conference on Research and Development in Information Retrieval*, pages 290–297, August 1998.

[68] N. Nisan. Algorithms for selfish agents. In *16th Annual Symposium on Theoretical Aspects of Computer Science*, March 1999.

[69] N. Nisan and A. Ronen. Algorithmic mechanism design. In *Proceedings of the ACM Symposium on Theory of Computing*, May 1999.

[70] OpenNap Home Page. http://opennap.sourceforge.net/.

[71] C. Palmer and J. Steffan. Generating network topologies that obey power laws. In *Proceedings of GLOBECOM*, November 2000.

[72] C. Papadimitriou. Algorithms, games, and the internet. In *Proceedings of the ACM Symposium on Theory of Computing*, July 2001.

[73] Michael Rabinovich, Jeff Chase, and Syan Gadde. Not all hits are created equal: Cooperative proxy caching over a wide area network. In *Proceedings of the Third International WWW Caching Workshop*, June 1998.

[74] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM International Conference on Data Communication*, August 2001.

[75] Webnoize Research. *Napster University: From File Swapping to the Future of Entertainment Commerce.* Webnoize, Inc., Cambridge, MA, 2000. http://www.webnoize.com/.

[76] B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the Third ACM Conference on Digital Libraries*, pages 182–190, June 1998.

[77] R. Rivest and A. Shamir. Payword and micromint: two simple micropayment schemes. In *Security Protocols Workshop*, 1996.

[78] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[79] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, October 2001.

[80] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 1995.

[81] G. Salton. *Information Retrieval: Data Structures and Algorithms.* Addison-Wesley, Massachussetts, 1989.

[82] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, January 2002.

[83] M. Schlosser, T. Condie, S. Kamvar, and H. Garcia-Molina. Simulating a file-sharing p2p network. In *First Workshop on Semantics in P2P and Grid Computing*, December 2002.

[84] J. Shneidman and D. Parkes. Rationality and self-interest in peer to peer networks. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems*, March 2003.

[85] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM International Conference on Data Communication*, August 2001.

[86] Q. Sun and H. Garcia-Molina. Slic: A selfish link-based incentive mechanism for unstructured p2p networks. In *Proceedings of the Twenty-Fourth International Conference on Distributed Computing*, 2004.

[87] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlays. In *Proceedings of the ACM SIGCOMM International Conference on Data Communication*, August 2003.

[88] Anthony Tomasic. *Distributed Queries and Incremental Updates in Information Retrieval Systems*. PhD thesis, Princeton University, 1994.

[89] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental update of inverted list for text document retrieval. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 289–300, May 1994.

[90] K. Truelove. To the bandwidth barrier and beyond. Originally an online document available through DSS Clip2, now unavailable because the company has gone out of business. Please contact the author of this thesis for a copy of the document.

[91] D. Tsoumakos and N. Roussopoulos. Adaptive probabilistic search for peer-to-peer networks. In *Proceedings of the Third Conference on P2P Computing*, September 2003.

[92] C. Wang, L. Xiao, Y. Liu, and P. Zhang. Distributed caching and adaptive search in multilayer p2p networks. In *Proceedings of the Twenty-Fourth International Conference on Distributed Computing Systems*, 2004.

[93] B. Yang, T. Condie, S. Kamvar, and H. Garcia-Molina. Non-cooperation in competitive p2p networks. Technical report, Stanford University, January 2004.

[94] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Proceedings of the Twenty-Seventh International Conference on Very Large Databases*, September 2001.

[95] B. Yang and H. Garcia-Molina. Improving efficiency of search in peer-to-peer networks. In *Proceedings of the Twenty-Second International Conference on Distributed Computing Systems*, July 2002.

[96] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proceedings of the Nineteenth International Conference on Data Engineering*, March 2003.

[97] B. Yang and H. Garcia-Molina. Ppay: Micropayments for peer-to-peer systems. In *Proceedings of the Tenth ACM Conference on Computer and Communications Security*, October 2003.

[98] B. Yang, P. Vinograd, and H. Garcia-Molina. Evaluating guess and non-forwarding peer-to-peer search. In *Proceedings of the Twenty-Fourth International Conference on Distributed Computing Systems*, March 2004.

[99] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001. Available at http://www.cs.berkeley.edu/ ravenben/publications/CSD-01-1141.pdf.

[100] S. Zhong, J. Chen, and Y. Yang. Sprite: A simple, cheat-proof, credit-based system for model ad-hoc networks. In *Proceedings of IEEE INFOCOM*, 2003.

[101] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proceedings of the Eighteenth International Conference on Very Large Databases*, pages 352–362, August 1992.